Microsoft Windows

FIPS 140 Validation

Microsoft Windows 11

Windows Server 2022

Microsoft Windows 10 (versions 20H2 and 21H1)

Microsoft Windows Server (version 20H2)

Windows Server Azure Edition

Azure Host 2021

Azure Stack HCI version 21H2

Azure Virtual Desktop

Non-Proprietary

Security Policy Document

Version Number	1.1
Updated On	June 24, 2024

Version History

Version	Date	Summary of changes
1.0	March 25, 2022	Draft sent to NIST CMVP
1.1	June 24, 2024	Updates in response to NIST feedback

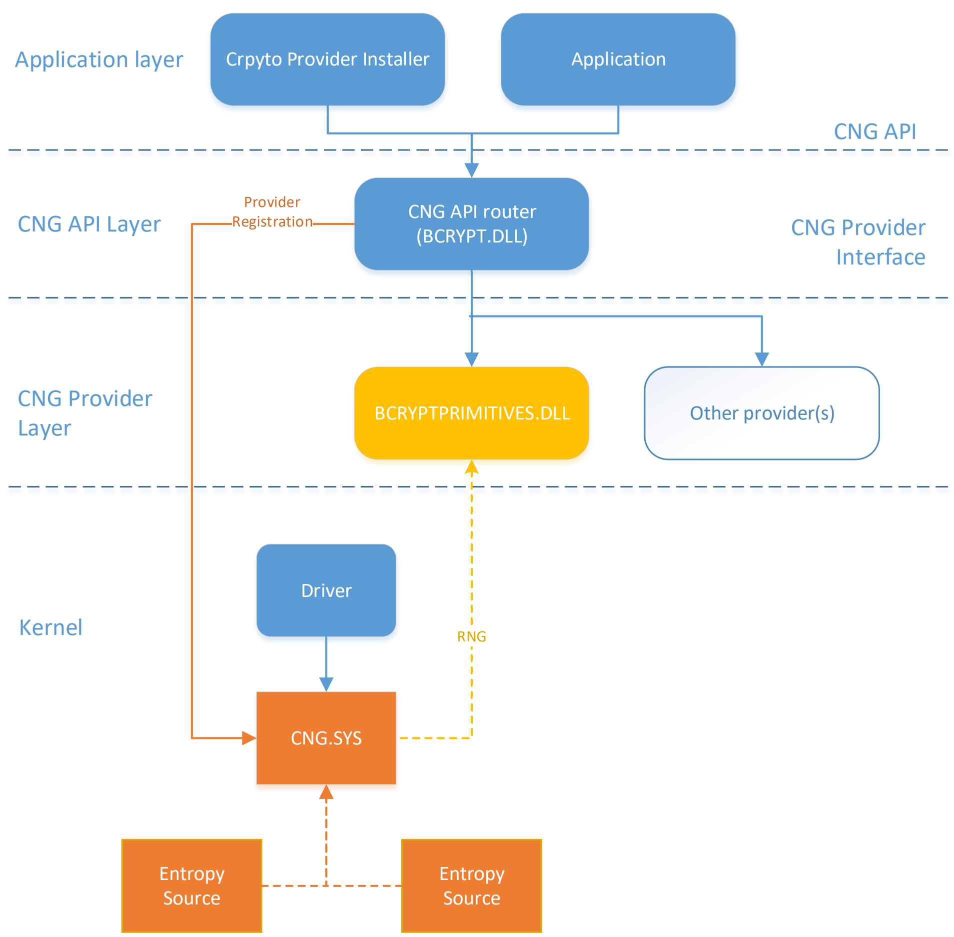
TABLE OF CONTENTS

1 Introduction

The Microsoft Kernel Mode Cryptographic Primitives Library is a kernel-mode cryptographic module that provides cryptographic services through the Microsoft CNG (Cryptography, Next Generation) API to Windows 10 kernel components.

The Kernel Mode Cryptographic Primitives Library also provides cryptographic provider registration and configuration services to both user and kernel mode components. See Non-Security Relevant Configuration Interfaces for more information.

The relationship between the Kernel Mode Cryptographic Primitives Library and other components is shown in the following diagram:



1.1 List of Cryptographic Module Binary Executables

The Kernel Mode Cryptographic Primitives Library consists of the following binary. Each binary has a distinct implementation per build.

- CNG.SYS

The Windows products covered by this validation are:

- Build 10.0.22000
  - Windows 11
- Build 10.0.20348
  - Windows Server 2022
  - Windows Server Azure Edition
  - Azure Host 2021
  - Azure Stack HCI version 21H2
- Build 10.0.19043:
  - Windows 10 version 21H1
- Build 10.0.19042
  - Windows 10 version 20H2
  - Windows Server version 20H2

1.2 Validated Platforms

The Windows editions covered by this validation are:

- Microsoft Windows 11
- Windows Server 2022
- Microsoft Windows 10 Pro Edition (64-bit version)
- Microsoft Windows 10 Enterprise Edition (64-bit version)
- Windows Server Core Standard
- Windows Server Core Datacenter
- Windows Server Azure Edition
- Azure Host 2021
- Azure Stack HCI

The Kernel Mode Cryptographic Primitives Library component listed in Section 1.1 was validated using the combination of computers and Windows operating system editions specified in the tables below.

All the computers for Windows 10 and Windows Server listed in the tables below are 64-bit Intel architecture and implement the AES-NI instruction set but not the SHA Extensions, with the following exception:

- HPE ProLiant E910 (Edgeline EL8000) – Intel Xeon Gold 6248, with AES-NI disabled and no SHA Extensions.

Table 1 Validated Platforms for Windows 10 and Windows Server version 20H2

Computer	Windows 10 Pro	Windows 10 Enterprise	Windows Server Core	Windows Server Core Datacenter
Microsoft Surface Laptop 4 - Intel i5-1145G7	√			
Microsoft Windows Server 2019 Hyper-V on Dell R630 - Intel Xeon E5-2660 v4			√	√
Dell Latitude 3520 - Intel i3-1115G4	√			
Dell Latitude 9520 - Intel i7-1185G7		√		
Dell Latitude 7420 - Intel i7-1185G7		√		
HP EliteBook x360 830 G8 - Intel i7-1165G7	√			

Table 2 Validated Platforms for Windows 10 version 21H1 and Windows Server 2022

Computer	Windows 10 Pro	Windows Server 2022 Core	Windows Server 2022 Core Datacenter
HPE ProLiant E910 (Edgeline EL8000) - Intel Xeon Gold 6248			√
Microsoft Surface Laptop 4 - Intel i5-1145G7	√		
Microsoft Windows Server 2019 Hyper-V on Dell R630 - Intel Xeon E5-2660 v4		√	√
HP EliteBook x360 830 G8 - Intel i7-1165G7	√		

Table 3 Validated Platforms for Windows 11 and Azure

Computer	Windows 11	Windows Server Azure Edition	Azure Host 2021	Azure Stack HCI version 21H2
Microsoft Surface Laptop 4 - Intel i5-1145G7	√			
Dell PowerEdge R840 - Intel Xeon Platinum 8260		√	√	
HPE ProLiant DL380 - Intel Xeon Platinum 8276L				√

1.3 Configure Windows to use FIPS-Approved Cryptographic Algorithms

There are two methods to enable FIPS-Approved mode for the Kernel Mode Cryptographic Primitives Library.

The first is to use FIPS Local/Group Security Policy setting or a Mobile Device Management (MDM) to enable FIPS-Approved mode for the Kernel Mode Cryptographic Primitives Library. The Windows operating system provides a group (or local) security policy setting, “System cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing”.

The second method to enable FIPS-Approved mode for the Kernel Mode Cryptographic Primitives Library is to set the following registry key to 1: HKLM\System\CurrentControlSet\Control\Lsa\FIPSAlgorithmPolicy\STE. When this registry key exists and is set to 1, the selftests in Kernel Mode Cryptogaphic Primitives Library will run in compliance with FIPS 140-2 Implementation Guidance section 9.11 and the module will be in FIPS Approved mode.

In addition to these methods, Consult the MDM documentation for information on how to enable FIPS-Approved mode. The Policy CSP - Cryptography includes the setting **AllowFipsAlgorithmPolicy**.

Changes to either Approved mode security policy setting do not take effect until the computer has been rebooted.

2 Cryptographic Module Specification

The Kernel Mode Cryptographic Primitives Library is a multi-chip standalone module that operates in FIPS-Approved mode during normal operation of the computer and Windows operating system and when Windows is configured to use FIPS-approved cryptographic algorithms as described in Configure Windows to use FIPS-Approved Cryptographic Algorithms.

In addition to configuring Windows to use FIPS-Approved Cryptographic Algorithms, third-party applications and drivers installed on the Windows platform must not use any of the non-Approved algorithms implemented by this module. Windows will not operate in an Approved mode when the operators chooses to use a non-Approved algorithm or service.

The following configurations and modes of operation will cause the Kernel Mode Cryptographic Primitives Library to operate in a non-Approved mode of operation:

- Boot Windows in Debug mode
- Boot Windows with Driver Signing disabled
- Windows enters the ACPI S4 power state

2.1 Cryptographic Boundary

The software cryptographic boundary for the Kernel Mode Cryptographic Primitives Library is defined as the binary CNG.SYS.

2.2 FIPS 140-2 Approved Algorithms

The Kernel Mode Cryptographic Primitives Library implements the following FIPS-140-2 Approved algorithms:[1]

Table 4 Algorithm Certificates for Windows 10, Windows Server, and Azure Virtual Desktop

Algorithm	Windows 10 and Windows Server version 20H2	Windows 10 version 21H1 and Azure Virtual Desktop version 21H1
FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512	#A2066	#A2025
FIPS PUB 198-1 HMAC-SHA-1[2], HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512	#A2066	#A2025
FIPS 197 AES-128, AES-192, and AES-256 in ECB, CBC, CFB8, CFB128, and CTR modes	#A2066	#A2025
NIST SP 800-38B and SP 800-38C AES-128, AES-192, and AES-256 in CCM and CMAC modes	#A2066	#A2025
NIST SP 800-38D AES-128, AES-192, and AES-256 GCM and GMAC	#A2066	#A2025
NIST SP 800-38E XTS-AES XTS-128 and XTS-256[3]	#A2066	#A2025

[1] This module may not use some of the capabilities described in each CAVP certificate. Only those algorithms/modes listed in the tables below are utilized by the module.
[2] For HMAC, only key sizes that are >= 112 bits in length are used by the module in FIPS mode.
[3] AES XTS must be used only to protect data at rest and the caller needs to ensure that the length of data encrypted does not exceed 2^{20} AES blocks.

Algorithm	Windows 10 and Windows Server version 20H2	Windows 10 version 21H1 and Azure Virtual Desktop version 21H1
FIPS 186-4 RSA PKCS#1 (v1.5) digital signature generation and verification with 1024, 2048, 3072, and 4096 moduli; supporting SHA-1[4], SHA-256, SHA-384, and SHA-512	#A2066	#A2025
Safe primes key generation with groups ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, MODP-2048, MODP-3072, and MODP-4096	#A2066	#A2025
FIPS 186-4 RSA key-pair generation with 2048 and 3072 moduli	#A2066	#A2025
FIPS 186-4 ECDSA key pair generation and verification, signature generation and verification with the following NIST curves: P-256, P-384, P-521	#A2066	#A2025
FIPS 186-4 DSA PQG generation and verification, signature generation and verification	#A2066	#A2025
NIST SP 800-56Arev3 KAS – Diffie-Hellman Key Agreement; Finite Field Cryptography (FFC) with domain parameters FB (p=2048, q=224), FC (p=2048, q=256), and safe primes (ffdhe2048, MODP-2048, ffdhe3072, MODP-3072, ffdhe4096, and MODP-4096); key establishment methodology provides at least 112 bits of encryption strength	#A2066	#A2025
NIST SP 800-56A rev3 KAS – EC Diffie-Hellman Key Agreement; Elliptic Curve Cryptography (ECC) with domain parameters 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	#A2066	#A2025
NIST SP 800-56A rev3 KAS-FFC-SSC key agreement (dhEphem, dhOneFlow, and dhStatic; KAS Roles: initiator, responder), with domain parameters FB, FC, and safe primes (ffdhe2048, MODP-2048)	#A2066	#A2025
NIST SP 800-56A rev3 KAS-ECC-SSC key agreement (ephemeralUnified; KAS roles: initiator, responder), with domain parameters P-256 (hash functions SHA2-256, SHA2-384, SHA2-512), P-384 (hash functions SHA2-384,	#A2066	#A2025

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

Algorithm	Windows 10 and Windows Server version 20H2	Windows 10 version 21H1 and Azure Virtual Desktop version 21H1
SHA2-512), and P-521 (hash function SHA2-512).		
NIST SP 800-56B RSADP (CVL) mod 2048	#A2066	#A2025
NIST SP 800-90A AES-256 counter mode DRBG	#A2066	#A2025
NIST SP 800-67r1 Triple-DES (2 key legacy-use decryption[5] and 3 key encryption/decryption) in ECB, CBC, CFB8 and CFB64 modes	#A2066	#A2025
NIST SP 800-108 Key Derivation Function (KBKDF) CMAC-AES (128, 192, 256), HMAC (SHA1, SHA-256, SHA-384, SHA-512)	#A2069	#A2031
NIST SP 800-38F AES Key Wrapping (KW) (128, 192, and 256), KTS (key establishment methodology provides between 128 and 256 bits of encryption strength)	#A2069	#A2031
NIST SP 800-135 IKEv1, IKEv2, TLS 1.0/1.1, and TLS 1.2 KDF primitive (CVL)[6]	#A2066	#A2025
NIST SP 800-132 KDF (also known as PBKDF) with HMAC (SHA-1, SHA-256, SHA-384, SHA-512) as the pseudo-random function	#A2066	#A2025
NIST SP 800-133 (Sections 5.1, 5.2, 6.1, and 6.2) Cryptographic Key Generation (CKG)	Vendor Affirmed	Vendor Affirmed

[5] 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. The caller is responsible for following the 2^16 guidelines in all uses.
[6] 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.

Algorithm	Windows 10 and Windows Server version 20H2	Windows 10 version 21H1 and Azure Virtual Desktop version 21H1
NIST SP 800-90B Entropy Source (ENT (P))	N/A	N/A
NIST SP 800-90B AES-CBC-MAC Conditioning Component	#A1791, #A2165, #A2138, #A2668	#A1791, #A2165, #A2138, #A2668

Table 5 Algorithm Certificates for Windows 11, Windows Server 2022, and Azure

Algorithm	Windows 11	Windows Server version 2022 and Windows Server Azure Edition	Azure Host 2021	Azure Stack HCI version 21H2
FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512	#A2004	#A2019	#A2019	#A2019
FIPS PUB 198-1 HMAC-SHA-1[7], HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512	#A2004	#A2019	#A2019	#A2019
FIPS 197 AES-128, AES-192, and AES-256 in ECB, CBC, CFB8, CFB128, and CTR modes	#A2004	#A2019	#A2019	#A2019
NIST SP 800-38B and SP 800-38C AES-128, AES-192, and AES-256 in CCM and CMAC modes	#A2004	#A2019	#A2019	#A2019
NIST SP 800-38D AES-128, AES-192, and AES-256 GCM and GMAC	#A2004	#A2019	#A2019	#A2019
NIST SP 800-38E XTS-AES XTS-128 and XTS-256[8]	#A2004	#A2019	#A2019	#A2019

[7] For HMAC, only key sizes that are >= 112 bits in length are used by the module in FIPS mode.

[8] AES XTS must be used only to protect data at rest and the caller needs to ensure that the length of data encrypted does not exceed 2^{20} AES blocks.

Algorithm	Windows 11	Windows Server version 2022 and Windows Server Azure Edition	Azure Host 2021	Azure Stack HCI version 21H2
FIPS 186-4 RSA PKCS#1 (v1.5) digital signature generation and verification with 1024, 2048, 3072, and 4096 moduli; supporting SHA-1[9], SHA-256, SHA-384, and SHA-512	#A2004	#A2019	#A2019	#A2019
Safe primes key generation with groups ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, MODP-2048, MODP-3072, MODP-4096	#A2004	#A2019	#A2019	#A2019
FIPS 186-4 RSA key-pair generation with 2048 and 3072 moduli	#A2004	#A2019	#A2019	#A2019
FIPS 186-4 ECDSA key pair generation and verification, signature generation and verification with the following NIST curves: P-256, P-384, P-521	#A2004	#A2019	#A2019	#A2019
FIPS 186-4 DSA PQG generation and verification, signature generation and verification	#A2004	#A2019	#A2019	#A2019
NIST SP 800-56Arev3 KAS – Diffie-Hellman Key Agreement; Finite Field Cryptography (FFC) with domain parameters FB (p=2048, q=224), FC (p=2048, q=256), and safe primes (ffdhe2048, MODP-2048, ffdhe3072, MODP-3072, ffdhe4096, and MODP-4096); key establishment methodology provides at	#A2004	#A2019	#A2019	#A2019

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

Algorithm	Windows 11	Windows Server version 2022 and Windows Server Azure Edition	Azure Host 2021	Azure Stack HCI version 21H2
least 112 bits of encryption strength				
NIST SP 800-56Arev3 KAS – EC Diffie-Hellman Key Agreement; Elliptic Curve Cryptography (ECC) with domain parameters 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	#A2004	#A2019	#A2019	#A2019
NIST SP 800-56A rev3 KAS-FFC-SSC key agreement (dhEphem, dhOneFlow, and dhStatic KAS Roles: initiator, responder), with domain parameters FB, FC, and safe primes (ffdhe2048, MODP-2048)	#A2004	#A2019	#A2019	#A2019
NIST SP 800-56B RSADP (CVL) mod 2048	#A2004	#A2019	#A2019	#A2019
NIST SP 800-90A AES-256 counter mode DRBG	#A2004	#A2019	#A2019	#A2019
NIST SP 800-67r1 Triple-DES (2 key legacy-use decryption[10] and 3 key encryption/decryption) in ECB, CBC, CFB8 and CFB64 modes	#A2004	#A2019	#A2019	#A2019

[10] 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. The caller is responsible for following the 2^16 guidelines in all uses.

Algorithm	Windows 11	Windows Server version 2022 and Windows Server Azure Edition	Azure Host 2021	Azure Stack HCI version 21H2
NIST SP 800-108 Key Derivation Function (KBKDF) CMAC-AES (128, 192, 256), HMAC (SHA1, SHA-256, SHA-384, SHA-512)	#A2001	#A2023	#A2023	#A2023
NIST SP 800-38F AES Key Wrapping (KW) (128, 192, and 256), KTS (key establishment methodology provides between 128 and 256 bits of encryption strength)	#A2001	#A2023	#A2023	#A2023
NIST SP 800-135 IKEv1, IKEv2, TLS 1.0/1.1, and TLS 1.2 KDF primitive (CVL)[11]	#A2004	#A2019	#A2019	#A2019
NIST SP 800-132 KDF (also known as PBKDF) with HMAC (SHA-1, SHA-256, SHA-384, SHA-512) as the pseudo-random function	#A2004	#A2019	#A2019	#A2019
NIST SP 800-133 (Sections 5.1, 5.2, 6.1, and 6.2) Cryptographic Key Generation (CKG)	Vendor Affirmed	Vendor Affirmed	Vendor Affirmed	Vendor Affirmed
NIST SP 800-90B Entropy Source (ENT (P))	N/A	N/A	N/A	N/A
NIST SP 800-90B AES-CBC-MAC Conditioning Component	#A1791, #A2165, #A2138, #A2668	#A1791, #A2165, #A2138, #A2668	#A1791, #A2165, #A2138, #A2668	#A1791, #A2165, #A2138, #A2668

[11] 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.

2.3 Non-Approved Algorithms

The Kernel Mode Cryptographic Primitives Library implements the following non-Approved but allowed algorithms:

- SHA-1 hash, which is disallowed for use in digital signature generation. It can be used for legacy digital signature verification. Its use is acceptable for non-digital signature generation applications.
- MD5 and HMAC-MD5 – allowed for TLS and EAP-TLS (no security claimed)
- KAS-ECC with the following curves that are allowed in FIPS mode as per FIPS 140-2 IG A.2

Curve	Security Strength (bits)	Allowed in FIPS mode
brainpoolP160r1	80	No
brainpoolP192r1	96	No
brainpoolP192t1	96	No
brainpoolP224r1	112	Yes
brainpoolP224t1	112	Yes
brainpoolP256r1	128	Yes
brainpoolP256t1	128	Yes
brainpoolP320r1	160	Yes
brainpoolP320t1	160	Yes
brainpoolP384r1	192	Yes
brainpoolP384t1	192	Yes
brainpoolP512r1	256	Yes
brainpoolP512t1	256	Yes
ec192wapi	96	No
nistP192	96	No
nistP224	112	Yes
numsP256t1	128	Yes
numsP384t1	192	Yes
numsP512t1	256	Yes
secP160k1	80	No
secP160r1	80	No
secP160r2	80	No
secP192k1	96	No
secP192r1	96	No
secP224k1	112	Yes
secP224r1	112	Yes
secP256k1	128	Yes
secP256r1	128	Yes
secP384r1	192	Yes
secP521r1	256	Yes
wtls12	112	Yes
wtls7	80	No
wtls9	80	No
x962P192v1	96	No
x962P192v2	96	No

Curve	Security Strength (bits)	Allowed in FIPS mode
x962P192v3	96	No
x962P239v1	120	Yes
x962P239v2	120	Yes
x962P239v3	120	Yes
x962P256v1	128	Yes

The Kernel Mode Cryptographic Primitives Library implements the following non-Approved algorithms but should not be used:

- Non-compliant HMAC. 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.
- RC2, RC4, MD2, MD4
- 2-Key Triple-DES Encryption, which is disallowed for usage altogether as of the end of 2015.
- DES in ECB, CBC, CFB8 and CFB64 modes
- Non-complaint RSA encrypt/decrypt
- Non-complaint IEEE 1619-2007 XTS-AES, XTS-128 and XTS-256
- Non-compliant AES GCM encryption except when the module operator does not follow the FIPS 140-2 Implementation Guidance A.5 scenario 4 for generating initialization vectors.
- Non-compliant RSA 1024-bits for digital signature generation, which is disallowed.
- Non-compliant FIPS 186-2 DSA with key length of 1024 bits
- Legacy CAPI KDF (proprietary)
- Non-complaint HKDF
- Non-compliant ANSI X9.63 and X9.42 key derivation
- NIST SP 800-56A Key Agreement using Finite Field Cryptography (FFC) with parameter FA (p=1024, q=160). The key establishment methodology provides 80 bits of encryption strength instead of the Approved 112 bits of encryption strength listed above.

2.4 FIPS 140-2 Approved Algorithms from Bounded Modules

A bounded module is a FIPS 140 module which provides cryptographic functionality that is relied on by a downstream module. As described in the Integrity Chain of Trust section, Code Integrity depends on the following modules and algorithms:

The Windows OS Loader (module certificate #4339) provides

- CAVP certificates #A2071 (Windows 10 and Windows Server version 20H2) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2066 (Windows 10 and Windows Server version 20H2) for FIPS 180-4 SHS SHA-256
- CAVP certificates #A2024 (Windows 10 version 21H1 Windows Server 2022) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2025 (Windows 10 version 20H1 and Windows Server 2022) for FIPS 180-4 SHS SHA-256

- CAVP certificates #A2003 (Windows 11) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2004 (Windows 11) for FIPS 180-4 SHS SHA-256
- CAVP certificates #A2018 (Microsoft Azure operating systems) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2019 (Microsoft Azure operating systems) for FIPS 180-4 SHS SHA-256

The Windows Resume (module certificate #4348) provides

- CAVP certificates #A2019 (Windows 10 and Windows Server) for NIST SP 800-38E AES XTS 128 and 256

The TCB Launcher module (module certificate #4457) provides:

- CAVP certificates #A2071 (Windows 10 and Windows Server version 20H2) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2066 (Windows 10 and Windows Server version 20H2) for FIPS 180-4 SHS SHA-256
- CAVP certificates #A2024 (Windows 10 version 21H1 Windows Server 2022) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2025 (Windows 10 version 20H1 and Windows Server 2022) for FIPS 180-4 SHS SHA-256
- CAVP certificates #A2003 (Windows 11) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2004 (Windows 11) for FIPS 180-4 SHS SHA-256
- CAVP certificates #A2018 (Microsoft Azure operating systems) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #A2019 (Microsoft Azure operating systems) for FIPS 180-4 SHS SHA-256

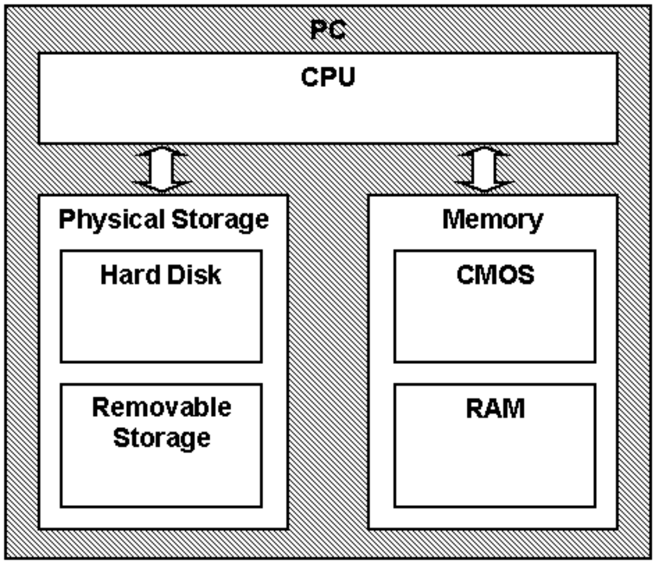
Note that the validated platforms listed in section 1.2 include processors that support AES-NI. This module does not implement AES, but the bounded modules may implement AES and, therefore, use AES-NI.

2.5 Cryptographic Bypass

Cryptographic bypass is not supported by Kernel Mode Cryptographic Primitives Library.

2.6 Hardware Components of the Cryptographic Module

The physical boundary of the module is the physical boundary of the computer that contains the module. The following diagram illustrates the hardware components of the Kernel Mode Cryptographic Primitives Library module:



3 Cryptographic Module Ports and Interfaces

The Kernel Mode 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 CNG API. For a full list of these algorithm providers, see https://docs.microsoft.com/en-us/windows/win32/seccng/cng-algorithm-identifiers

The Kernel Mode Cryptographic Primitives Library module is accessed through one of the following logical interfaces:

1. Kernel applications requiring cryptographic services use the BCrypt APIs detailed in Services.
2. Entropy sources supply random bits to the random number generator through the entropy interfaces.

3.1 CNG Primitive Functions

The following security-relevant functions are exported by the Kernel Mode 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
- SystemPrng
- EntropyPoolTriggerReseedForIum
- EntropyProvideData
- EntropyRegisterSource
- EntropyUnregisterSource

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

The Kernel Mode Cryptographic Primitives Library has additional export functions described in Non-Security Relevant Configuration Interfaces.

3.1.1 Algorithm Providers and Properties

3.1.1.1 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.

Unless the calling function specifies the name of the provider, the default provider is used.

The calling function must pass the BCRYPT_ALG_HANDLE_HMAC_FLAG flag in order to use an HMAC function with a hash algorithm.

3.1.1.2 BCryptCloseAlgorithmProvider

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

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

3.1.1.3 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. The CNG object is a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

3.1.1.4 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. The CNG object is a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

3.1.1.5 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.

3.1.2 Random Number Generation

3.1.2.1 BCryptGenRandom

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

The BCryptGenRandom() function fills a buffer with random bytes. The random number generation algorithm is:

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

This function is a wrapper for SystemPrng.

3.1.2.2 SystemPrng

```
BOOL SystemPrng(
        unsigned char   *pbRandomData,
        size_t        cbRandomData );
```

The SystemPrng() function fills a buffer with random bytes generated from output of NIST SP 800-90A AES-256 counter mode based DRBG seeded from the Windows entropy pool. The Windows entropy pool is populated from the following sources:

- An initial entropy value provided by the Windows OS Loader at boot time.
- The values of the high-resolution CPU cycle counter at times when hardware interrupts are received.
- Random values gathered from the Trusted Platform Module (TPM), if one is available on the system.
- Random values gathered by calling the RDRAND CPU instruction, if supported by the CPU.

The Windows DRBG infrastructure located in cng.sys continues to gather entropy from these sources during normal operation, and the DRBG cascade is periodically reseeded with new entropy.

3.1.2.3 EntropyRegisterSource

```
NTSTATUS EntropyRegisterSource(
        ENTROPY_SOURCE_HANDLE * phEntropySource,
        ENTROPY_SOURCE_TYPE     entropySourceType,
    PCWSTR              entropySourceName );
```

This function is used to obtain a handle that can be used to contribute randomness to the Windows entropy pool. The handle is returned in the phEntropySource parameter. For this function, entropySource must be set to ENTROPY_SOURCE_TYPE_HIGH_PUSH, and entropySourceName must be a Unicode string describing the entropy source.

3.1.2.4 EntropyUnregisterSource

```
NTSTATUS EntropyRegisterSource(
ENTROPY_SOURCE_HANDLE hEntropySource);
```

This function is used to destroy a handle created with EntropyRegisterSource().

3.1.2.5 EntropyProvideData

```
NTSTATUS EntropyProvideData(
    ENTROPY_SOURCE_HANDLE   hEntropySource,
    PCBYTE              pbData,
    SIZE_T              cbData,
    ULONG               entropyEstimateInMilliBits );
```

This function is used to contribute entropy to the Windows entropy pool. hEntropySource must be a handle returned by an earlier call to EntropyRegisterSource. The caller provides cbData bytes in the buffer pointed to by pbData, as well as an estimate (in the entropyEstimateInMilliBits parameter) of how many millibits of entropy are contained in these bytes.

3.1.2.6 EntropyPoolTriggerReseedForIum

```
VOID EntropyPoolTriggerReseedForIum(BOOLEAN fPerformCallbacks);
```

This function will trigger a kernel DRBG reseed for the cng.sys inside the IUM (Isolated User Mode) environment. If called inside the IUM environment, it triggers a reseed from one or more of the entropy pools of the system. If called inside the normal world (non-IUM) environment, this function does nothing.

3.1.3 Key and Key-Pair Generation

3.1.3.1 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 directly from a DRBG for use with a symmetric encryption algorithm or key derivation algorithm from a supplied key value. The calling application must specify a handle to the algorithm provider created with the BCryptOpenAlgorithmProvider() function. The algorithm specified when the provider was created must support symmetric key encryption or key derivation.

3.1.3.2 BCryptGenerateKeyPair

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

The BCryptGenerateKeyPair() function creates an empty public/private key pair. After creating a key using this function, call the BCryptSetProperty() function to set its properties. The key pair can be used only after BCryptFinalizeKeyPair() function is called.

3.1.3.3 BCryptFinalizeKeyPair

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

The BCryptFinalizeKeyPair() function completes a public/private key pair import or generation directly from the output of a DRBG. 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.

3.1.3.4 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.

3.1.3.5 BCryptDestroyKey

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

The BCryptDestroyKey() function destroys the specified key.

3.1.4 Key Entry and Output

3.1.4.1 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.

3.1.4.2 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.

3.1.4.3 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.

3.1.5 Encryption and Decryption

3.1.5.1 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.

3.1.5.2 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.

3.1.6 Hashing and Message Authentication

3.1.6.1 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.

3.1.6.2 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.

3.1.6.3 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.

3.1.6.4 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.

3.1.6.5 BCryptDestroyHash

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

The BCryptDestroyHash() function destroys a hash object.

3.1.6.6 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.

3.1.6.7 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.

3.1.6.8 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.

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.

3.1.7 Signing and Verification

3.1.7.1 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.
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.

3.1.7.2 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. 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.

3.1.8 Secret Agreement and Key Derivation

3.1.8.1 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 KAS-FFC and KAS-ECC algorithms.

3.1.8.2 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.
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.800-56Ar3.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 bits for EC, 192 for ED, 256 for EE.

3.1.8.3 BCryptDestroySecret

```
NTSTATUS WINAPI BCryptDestroySecret(
        BCRYPT_SECRET_HANDLE   hSecret);
```

The BCryptDestroySecret() function destroys a secret agreement handle that was created by using the BCryptSecretAgreement() function.

3.1.8.4 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.

3.1.8.5 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).

3.1.9 Cryptographic Transitions

3.1.9.1 KAS-FFC and KAS-ECC

Through the year 2010, implementations of KAS-FFC and KAS-ECC were allowed to have an acceptable bit strength of at least 80 bits of security (for KAS-FFC at least 1024 bits and for KAS-ECC 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. KAS-ECC uses curve sizes of at least 256 bits (that means it has at least 128 bits of security strength), so that is acceptable. However, KAS-FFC has a range of 1024 to 4096 and that changed to 2048 to 4096 after 2013.

3.1.9.2 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.

3.2 Control Input Interface

The Control Input Interface are the functions in Algorithm Providers and Properties. Options for control operations are passed as input parameters to these functions.

3.3 Status Output Interface

The Status Output Interface for the Kernel Mode Cryptographic Primitives Library is the return value from each export function in the Kernel Mode Cryptographic Primitives Library.

3.4 Data Output Interface

The Data Output Interface for the Kernel Mode Cryptographic Primitives Library consists of the Kernel Mode Cryptographic Primitives Library export functions except for the Control Input Interfaces. Data is returned to the function's caller via output parameters.

3.5 Data Input Interface

The Data Input Interface for the Kernel Mode Cryptographic Primitives Library consists of the Kernel Mode Cryptographic Primitives Library export functions except for the Control Input Interfaces. Data and options are passed to the interface as input parameters to the export functions. Data Input is kept separate from Control Input by passing Data Input in separate parameters from Control Input.

3.6 Non-Security Relevant Configuration Interfaces

The following interfaces are not cryptographic functions and are used to configure cryptographic providers on the system. Please see https://msdn.microsoft.com for details.

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 **CngGetFipsAlgorithmMode**	Determines whether the Kernel Mode 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.
EntropyRegisterCallback	Registers the callback function that will be called in a worker thread after every reseed that the system performs. The callback is merely informational.
BCryptQueryProviderRegistration	Retrieves information about a CNG provider.
BCryptEnumRegisteredProviders	Retrieves information about the registered providers.
BCryptCreateContext	Creates a new CNG configuration context.
BCryptDeleteContext	Deletes an existing CNG configuration context.
BCryptEnumContexts	Obtains the identifiers of the contexts in the specified configuration table.
BCryptConfigureContext	Sets the configuration information for an existing CNG context.
BCryptQueryContextConfiguration	Retrieves the current configuration for the specified CNG context.
BCryptAddContextFunction	Adds a cryptographic function to the list of functions that are supported by an existing CNG context.
BCryptRemoveContextFunction	Removes a cryptographic function from the list of functions that are supported by an existing CNG context.
BCryptEnumContextFunctions	Obtains the cryptographic functions for a context in the specified configuration table.
BCryptConfigureContextFunction	Sets the configuration information for the cryptographic function of an existing CNG context.
BCryptQueryContextFunctionConfiguration	Obtains the cryptographic function configuration information for an existing CNG context.
BCryptEnumContextFunctionProviders	Obtains the providers for the cryptographic functions for a context in the specified configuration table.
BCryptSetContextFunctionProperty	Sets the value of a named property or a cryptographic function in an existing CNG context.
BCryptQueryContextFunctionProperty	Obtains the value of a named property for a cryptographic function in an existing CNG context.
BCryptSetAuditingInterface	Sets the auditing interface.

4 Roles, Services and Authentication

4.1 Roles

The Kernel Mode Cryptographic Primitives Library is a kernel-mode driver that does not interact with the user through any service therefore the module's functions are fully automatic and not configurable. FIPS 140 validations define formal "User" and "Cryptographic Officer" roles. Both roles can use any of this module's services.

4.2 Services

The Kernel Mode Cryptographic Primitives Library services are:

1. **Algorithm Providers and Properties** – This module provides interfaces to register algorithm providers
2. **Random Number Generation**
3. **Key and Key-Pair Generation**
4. **Key Entry and Output**
5. **Encryption and Decryption**
6. **Hashing and Message Authentication**
7. **Signing and Verification**
8. **Secret Agreement and Key Derivation**
9. **Show Status**
10. **Self-Tests** - The module provides a power-up self-tests service that is automatically executed when the module is loaded into memory. See Self-Tests.
11. **Zeroizing Cryptographic Material -** See Cryptographic Key Management

4.2.1 Mapping of Services, Algorithms, and Critical Security Parameters

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

Service	Algorithms	CSPs
Algorithm Providers and Properties	None	None
Random Number Generation	AES-256 CTR DRBG ENT (P)	AES-CTR DRBG Seed AES-CTR DRBG Entropy Input AES-CTR DRBG V AES-CTR DRBG Key
Key and Key-Pair Generation	RSA, KAS-FFC, KAS-ECC, 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, GCM[12], and GMAC modes; • NIST SP XTS-AES XTS-128 and XTS-256; • SP 800-56B RSADP mod 2048 (AES-GCM encryption[13], IEEE 1619-2007 XTS-AES, RC2, RC4, RSA, and DES, which cannot be used in FIPS mode)	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; • MD5 and HMAC-MD5 (allowed in TLS and EAP-TLS); • MD2 and MD4 (disallowed in FIPS mode)	Symmetric Keys (for HMAC, AES CCM, AES CMAC, and AES GMAC)
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[14], SHA-256, SHA-384, and SHA-512 • FIPS 186-4 ECDSA with the following NIST curves: P-256, P-384, P-521	Asymmetric Public Keys Asymmetric RSA Private Keys Asymmetric ECDSA Public Keys Asymmetric ECDSA Private keys
Secret Agreement and Key Derivation	• KAS-FFC – SP 800-56Arev3 Diffie-Hellman Key Agreement, Finite Field	DH Private and Public Values,

[12] If the initialization vector was not generated according to IG A.5 Scenario 4, refer to section 7.3 Key Generation for additional information about generating IVs.
[13] Idem.
[14] SHA-1 is only acceptable for legacy signature verification.

	Cryptography (FFC); 2048-4096 bit key size • KAS-ECC – SP 800-56Arev3 EC Diffie-Hellman Key Agreement with the following NIST curves: P-256, P-384, P-521 and the FIPS non-Approved curves listed in Non-Approved Algorithms • SP 800-56A rev3 KAS-FFC-SSC key agreement (dhEphem, dhOneFlow, and dhStatic; KAS Roles: initiator, responder), with domain parameters FB, FC, and safe primes (ffdhe2048, MODP-2048) • SP 800-56A rev3 KAS-ECC-SSC key agreement (ephemeralUnified; KAS roles: initiator, responder), with domain parameters P-256 (hash functions SHA2-256, SHA2-384, SHA2-512), P-384 (hash functions SHA2-384, SHA2-512), and P-521 (hash function SHA2-512) • SP 800-108 Key Derivation Function (KDF) CMAC-AES (128, 192, 256), HMAC (SHA1, SHA-256, SHA-384, SHA-512) • SP 800-132 PBKDF • SP 800-135 IKEv1 and IKEv2 KDF primitives • Legacy CAPI KDF (cannot be used in FIPS mode) • HKDF (cannot be used in FIPS mode)	ECDH Private and Public Values, Z, Key Derivation Key, and TLS Pre-Master Secret
Show Status	None	None
Self-Tests	See Section 8 Self-Tests for the list of algorithms	None
Zeroizing Cryptographic Material	None	None

4.2.2 Mapping of Services, Export Functions, and Invocations

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

Service	Export Functions	Invocations
Algorithm Providers and Properties	BCryptOpenAlgorithmProvider BCryptCloseAlgorithmProvider BCryptSetProperty BCryptGetProperty BCryptFreeBuffer	This service is executed whenever one of these exported functions is called.
Random Number Generation	BcryptGenRandom SystemPrng EntropyRegisterSource EntropyUnregisterSource EntropyProvideData EntropyPoolTriggerReseedForIum	This service is executed whenever one of these exported functions is called.
Key and Key-Pair Generation	BCryptGenerateSymmetricKey BCryptGenerateKeyPair BCryptFinalizeKeyPair BCryptDuplicateKey BCryptDestroyKey	This service is executed whenever one of these exported functions is called.
Key Entry and Output	BCryptImportKey BCryptImportKeyPair BCryptExportKey	This service is executed whenever one of these exported functions is called.
Encryption and Decryption	BCryptEncrypt BCryptDecrypt	This service is executed whenever one of these exported functions is called.
Hashing and Message Authentication	BCryptCreateHash BCryptHashData BCryptDuplicateHash BCryptFinishHash BCryptDestroyHash BCryptHash BCryptCreateMultiHash BCryptProcessMultiOperations	This service is executed whenever one of these exported functions is called.
Signing and Verification	BCryptSignHash BCryptVerifySignature	This service is executed whenever one of these exported functions is called.
Secret Agreement and Key Derivation	BCryptSecretAgreement BCryptDeriveKey BCryptDestroySecret BCryptKeyDerivation BCryptDeriveKeyPBKDF2	This service is executed whenever one of these exported functions is called.
Show Status	All Exported Functions	This service is executed upon completion of an exported function.
Self-Tests	DriverEntry	This service is executed upon startup of this module.

Zeroizing Cryptographic Material	BCryptDestroyKey BCryptDestroySecret	This service is executed whenever one of these exported functions is called.

4.2.3 Non-Approved Services

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

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.
BCRYPT_KDF_HKDF	Derives a key from a hash value. This function is provided to support potential enhancements to Windows.
SslDecryptPacket **SslEncryptPacket** **SslExportKey** **SslFreeObject** **SslImportKey** **SslLookupCipherLengths** **SslLookupCipherSuiteInfo** **SslOpenProvider** **SslIncrementProviderReferenceCount** **SslDecrementProviderReferenceCount**	Supports Secure Sockets Layer (SSL) protocol functionality. These functions are non-approved.

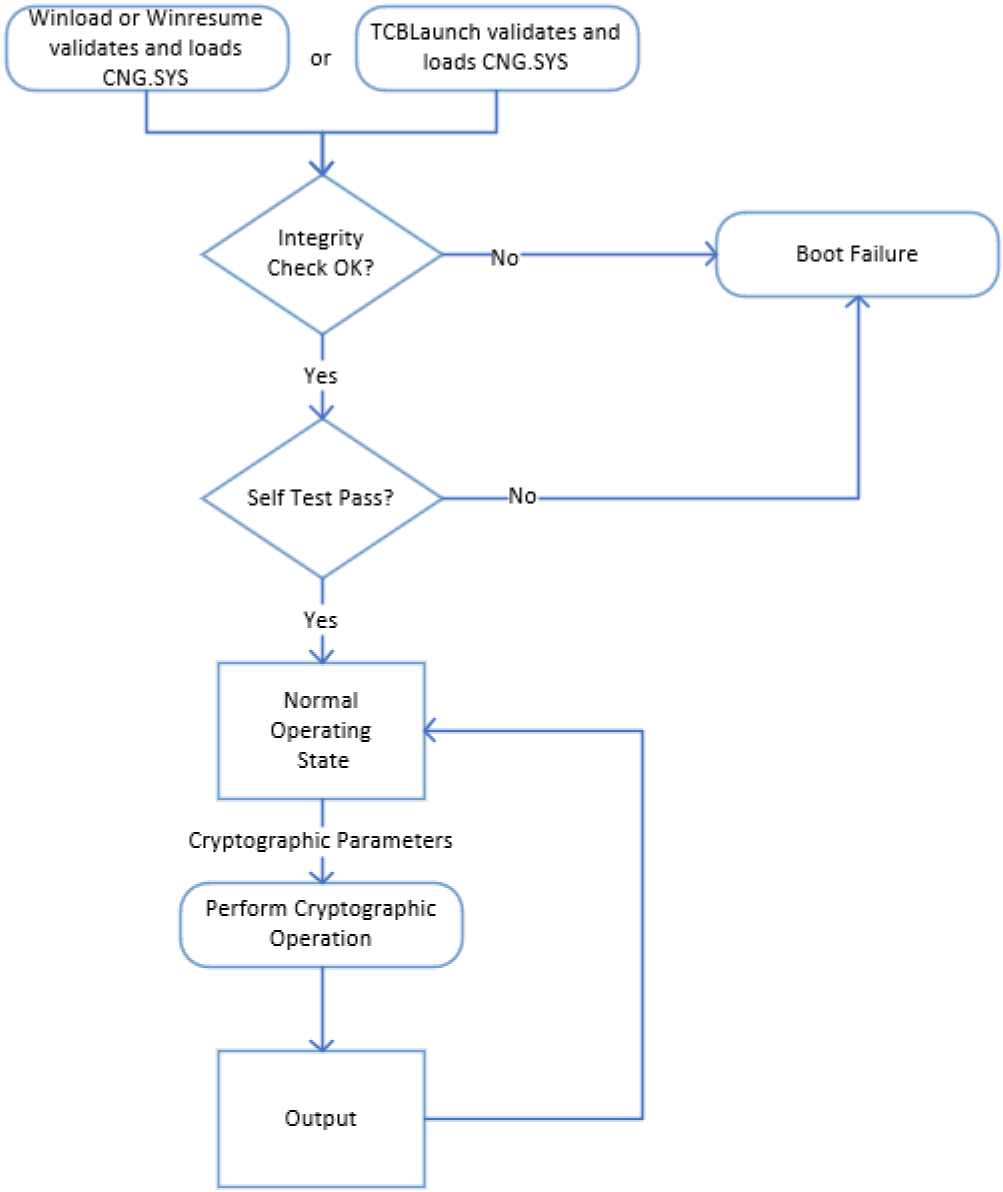
4.3 Authentication

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

5 Finite State Model

5.1 Specification

The following diagram shows the finite state model for the Kernel Mode Cryptographic Primitives Library:



6 Operational Environment

The operational environment for the Kernel Mode Cryptographic Primitives Library is the Windows 10 operating system running on a supported hardware platform.

6.1 Single Operator

The Kernel Mode Cryptographic Primitives Library is loaded into kernel memory as part of the boot process and before the logon component is initialized. The “single operator” for the module is the Windows Kernel.

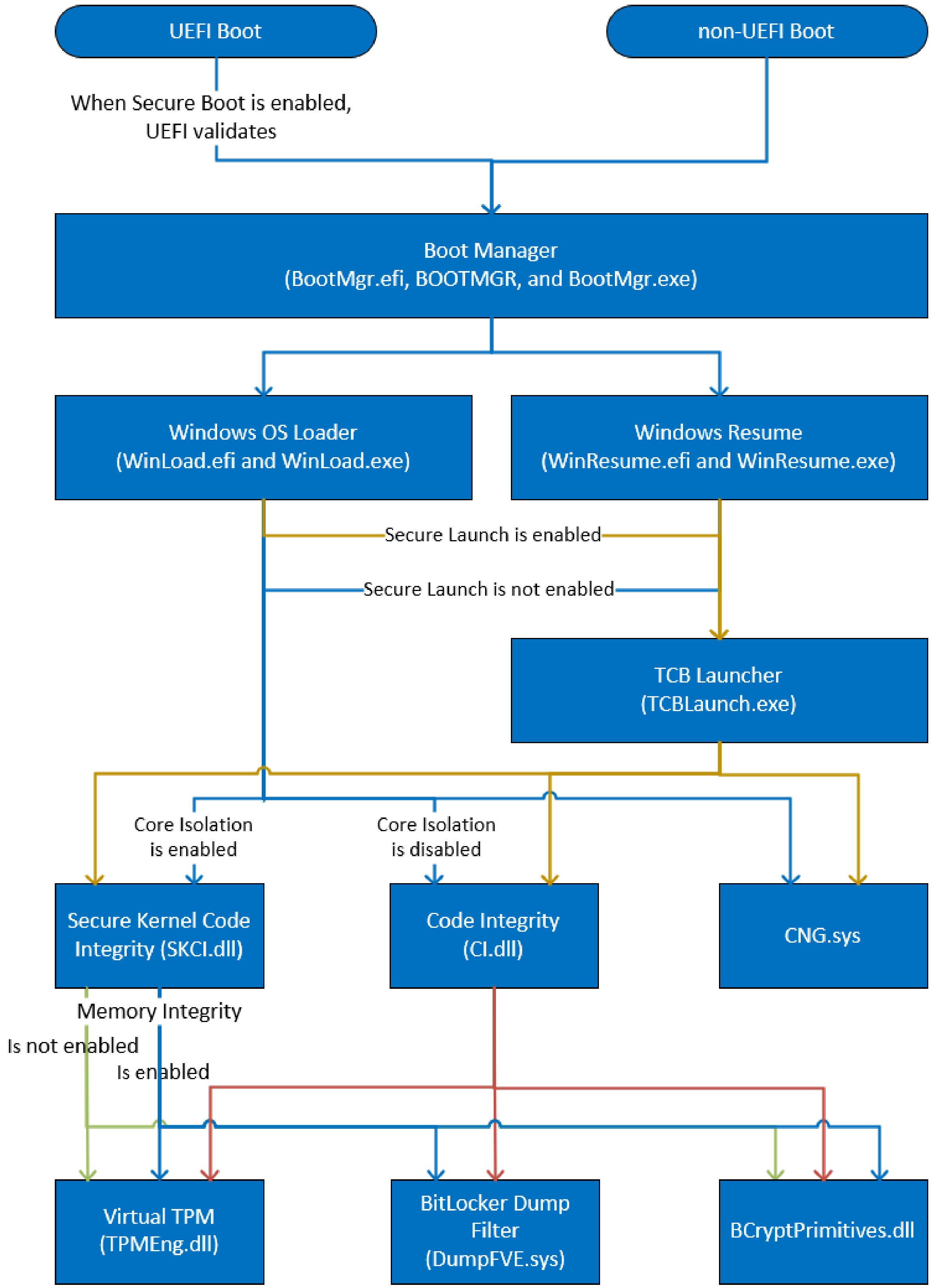
6.2 Cryptographic Isolation

In the Windows operating system, all kernel-mode modules, including CNG.SYS, are loaded into the Windows Kernel (ntoskrnl.exe) which executes as a single process. The Windows operating system environment enforces process isolation from user-mode processes including memory and CPU scheduling between the kernel and user-mode processes.

6.3 Integrity Chain of Trust

Windows uses several mechanisms to provide integrity verification depending on the stage in the boot sequence and also on the hardware and configuration. The following diagram describes the Integrity Chain of trust for each supported configuration for the following versions:

- Windows 11 build 10.0.22000
- Windows Server 2022 build 10.0.20348
- Windows 10 version 20H2 build 10.0.19042
- Windows Server version 20H2 build 10.0.19042
- Windows 10 version 21H1 build 10.0.19043
- Windows Server Azure Edition build 10.0.20348
- Azure Host 2021 build 10.0.20348
- Azure Stack HCI version 21H2 build 10.0.20348
- Azure Virtual Desktop version 21H1 build 10.0.19043


UEFI Boot
non-UEFI Boot
When Secure Boot is enabled, UEFI validates
Boot Manager
(BootMgr.efi, BOOTMGR, and BootMgr.exe)
Windows OS Loader
(WinLoad.efi and WinLoad.exe)
Windows Resume
(WinResume.efi and WinResume.exe)
Secure Launch is enabled
Secure Launch is not enabled
TCB Launcher
(TCBLaunch.exe)
Core Isolation is enabled
Core Isolation is disabled
Secure Kernel Code Integrity (SKCI.dll)
Code Integrity (CI.dll)
CNG.sys
Memory Integrity
Is not enabled
Is enabled
Virtual TPM
(TPMEng.dll)
BitLocker Dump Filter
(DumpFVE.sys)
BCryptPrimitives.dll

Note: TCB Launcher was not tested for Windows 10 version 1903

The integrity of the the Kernel Mode Cryptographic Primitives Library module is checked by the Windows OS Loader, Windows Resume, or TCB Launcher before it is loaded into ntoskrnl.exe.

Windows binaries include a SHA-256 hash of the binary signed with the 2048 bit Microsoft RSA code-signing key (i.e., the key associated with the Microsoft code-signing certificate). The integrity check uses the public key component of the Microsoft code signing certificate to verify the signed hash of the binary.

7 Cryptographic Key Management

The Kernel Mode Cryptographic Primitives Library module uses the following critical security parameters (CSPs) for FIPS Approved security functions:

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; key length: 112-2048, increment 8.
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 Entropy Input	A secret value that is at least 256 bits and maintained internal to the module that provides the entropy material for AES-CTR DRBG output[15]
AES-CTR DRBG Seed	A 384 bit secret value maintained internal to the module that provides the seed material for AES-CTR DRBG output[16]
AES-CTR DRBG V	A 128 bit secret value maintained internal to the module that provides the entropy material for AES-CTR DRBG output[17]

[15] Microsoft Common Criteria Windows Security Target, Page 29.
[16] Recommendation for Random Number Generation Using Deterministic Random Bit Generators, NIST SP 800-90A Revision 1, page 49.
[17] Ibid.

AES-CTR DRBG Key	A 256 bit secret value maintained internal to the module that provides the entropy material for AES-CTR DRBG output[18]
DH Private and Public values	Private and public values used for KAS-FFC key establishment. Key sizes are 2048 to 4096 bits.
ECDH Private and Public values	Private and public values used for KAS-ECC key establishment. Curve sizes are P-256, P-384, and P-521 and the ones listed in section 2.3.
Z	Shared secret input for KDFs and shared secret calculation output for SP 800-56Ar3 key agreement. Key size for KAS-FFC is 2048-4096 bits (input key size 2048, 3072, or 4096 bits); curves for KAS-ECC include P-256, P-384, and P-521 (input key size 256, 384, or 521 bits).
Key Derivation Key	Internal key for two-step KDFs. 256 or 384 bits.
TLS Pre-Master Secret	Shared secret input to the TLS KDF. Input size is dependent on the key exchange method of the chosen TLS cipher suite: for TLS_ECDHE_*, see the ECDH curve sizes listed above; for TLS_DHE_*, see the DH key sizes listed above; for TLS_RSA_*, the pre-master secret size is 384 bits.

7.1 Access Control Policy

The Kernel Mode Cryptographic Primitives Library module allows controlled access to the 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.

Kernel Mode Cryptographic Primitives Library crypto module **Service Access Policy**	Symmetric encryption/decryption keys	HMAC keys	Asymmetric ECDSA Public keys	Asymmetric ECDSA Private keys	Asymmetric RSA Public Keys	Asymmetric RSA Private Keys	AES-CTR DRBG Seed, AES-CTR DRBG Entropy Input, AES-CTR DRBG V, & AES-CTR DRBG key	DH Public and Private values	ECDH Public and Private values	Z	Key Derivation Key	TLS Pre-Master Secret
Algorithm Providers and Properties												
Random Number Generation							x					
Key and Key-Pair Generation	wd	wd	wd	wd	wd	wd	x	w d	wd			

18 Ibid.

Key Entry and Output	rw	rw	rw	rw	rw	rw		rw	rw			
Encryption and Decryption	x											
Hashing and Message Authentication		wx										
Signing and Verification			x	x	x	x	x					
Secret Agreement and Key Derivation							x	x	x	rw	rw	r
Show Status												
Self-Tests												
Zeroizing Cryptographic Material	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd

7.2 Key Material

When the Kernel Mode Cryptographic Primitives Library is loaded in the Windows 10 operating system kernel, no keys exist within it. A kernel module is responsible for importing keys into the Kernel Mode Cryptographic Primitives Library or using the Kernel Mode Cryptographic Primitives Library's functions to generate keys.

7.3 Key Generation

The Kernel Mode Cryptographic Primitives Library can create and use keys for the following algorithms: RSA, KAS-FFC, KAS-ECC, 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, and AES keys. When the operator chooses to have this cryptographic module generate initialization vectors for AES GCM mode in accordance with FIPS 140-2 Implementation Guidance A.5 scenario 4, then the call BCryptGenerateSymmetricKey() must set dwFlags to 0x00000020.

Asymmetric key-pairs are generated following the techniques given in SP 800-56Arev3 (Section 5.8). RSA and ECDSA keys and key-pairs are generated following the techniques given in FIPS 186-4. KAS-FFC and KAS-ECC keys and key-pairs are generated following the techniques given in SP 800-56Arev3.

Keys generated while not operating in the FIPS mode of operation cannot be used in FIPS mode, and vice versa.

7.4 Key Establishment

The Kernel Mode Cryptographic Primitives Library can use FIPS-Approved KAS-FFC and KAS-ECC key agreement, 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.

The Kernel Mode Cryptographic Primitives Library can use the following FIPS-Approved key derivation functions (KDF) from the common secret that is established during the execution of KAS-FFC and KAS-ECC key agreement algorithms:

- BCRYPT_KDF_SP80056A_CONCAT. This KDF supports the Concatenation KDF as specified in SP 800-56Arev3 (Section 5.8).
- 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.

The Kernel Mode Cryptographic Primitives Library can use the following FIPS-Approved key derivation functions (KDF) from a specified secret or password:

- BCRYPT_SP80056A_CONCAT_ALGORITHM. This KDF supports the Concatenation KDF as specified in SP 800-56Arev3 (Section 5.8).
- BCRYPT_SP800108_CTR_HMAC_ALGORITHM. This KDF supports the counter-mode variant of the KDF specified in SP 800-108r1 (Section 4.1) with HMAC as the underlying PRF.
- BCRYPT_PBKDF2_ALGORITHM. This KDF supports the Password Based Key Derivation Function specified in SP 800-132 (Section 5.3).

In addition, the industry standard KDF, HKDF (CNG flag BCRYPT_KDF_HKDF), and the legacy proprietary CryptDerive Key KDF, (BCRYPT_CAPI_KDF_ALGORITHM, described at https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptderivekey). Cannot be used in a FIPS-Approved mode.

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

There are two 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. The 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 described in SP 800-132, may only be used for 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 when the password/passphrase is created and not by this cryptographic module.[19]

7.4.2 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.

7.5 Key Entry and Output

Keys can be both exported and imported out of and into the Kernel Mode 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.

7.6 Key Storage

The Kernel Mode Cryptographic Primitives Library does not provide persistent storage of keys.

7.7 Key Archival

The Kernel Mode Cryptographic Primitives Library does not directly archive cryptographic keys. A 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. All key copies inside the Kernel Mode Cryptographic Primitives Library are destroyed and their memory location zeroized after used. It is the caller's responsibility to maintain the security of keys when the keys are outside the Kernel Mode Cryptographic Primitives Library.

7.8 Key Zeroization

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

[19] The probability of guessing a password is determined by its length and complexity, an organization should define a policy for these based based their threat model, suh as the example guidance in NIST SP800-63b, Appendix A.

8 Self-Tests

8.1 Power-On Self-Tests

The Kernel Mode Cryptographic Primitives Library module implements Known Answer Test (KAT) functions when the module is loaded into ntoskrnl.exe at boot time and the default driver entry point, DriverEntry, is called.

The Kernel Mode Cryptographic Primitives Library performs the following power-on (startup) self-tests:

- 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
- ECDSA sign/verify Known Answer Tests on P256 curve
- KAS-FFC secret agreement Known Answer Test with 2048-bit key
- KAS-ECC secret agreement Known Answer Test on P256 curve
- SP 800-90A AES-256 counter mode DRBG Known Answer Tests (instantiate, generate and reseed)
- SP800-90B startup health tests (APT/RCT)
- SP 800-108 KDF Known Answer Test
- SP 800-132 PBKDF Known Answer Test
- SHA-256 Known Answer Test
- SHA-512 Known Answer Test
- SP800-135 TLS 1.0/1.1 KDF Known Answer Test
- SP800-135 TLS 1.2 KDF Known Answer Test
- IKE SP800_135 KDF Known Answer Test

In any self-test fails, the Kernel Mode Cryptographic Primitives Library module does not load, an error code is returned to ntoskrnl.exe, and the computer will fail to boot.

8.2 Conditional Self-Tests

The Kernel Mode Cryptographic Primitives Library performs pair-wise consistency checks upon each invocation of RSA, KAS-ECC, and ECDSA key-pair generation and import as defined in FIPS 140-2.

KAS-FFC and KAS-ECC key usage assurances are performed according to NIST SP 800-56Arev3 sections 5.5.2, 5.6.2, and 5.6.3.

A Continuous Random Number Generator Test (CRNGT) and the DRBG health tests are performed for SP 800-90A AES-256 CTR DRBG.

The Entropy Source conducts Adaptive Proportion (APT) and Repetition Count (RCT) tests according to SP 800-90B.

When BCRYPT_ENABLE_INCOMPATIBLE_FIPS_CHECKS flag (required by policy) is used with BCryptGenerateSymmetricKey, then the XTS-AES Key_1 ≠ Key_2 check is performed in compliance with FIPS 140-2 IG A.9.

If the conditional self-test fails the function returns the status code STATUS_INTERNAL_ERROR.

9 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 operating system.

The Windows 10 operating system must be pre-installed on a computer by an OEM, installed by the end-user, by an organization's IT administrator, or updated from a previous Windows 10 version downloaded from Windows Update.

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 must be verified to match the version that was validated. See Appendix A – How to Verify Windows Versions and Digital Signatures 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 – How to Verify Windows Versions and Digital Signatures for details on how to do this.

10 Mitigation of Other Attacks

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

Algorithm	Protected Against	Mitigation
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

Algorithm	Protected Against	Mitigation
	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

11 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

12 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://docs.microsoft.com/en-us/windows/security/threat-protection/fips-140-validation

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

13.1 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.

13.2 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.

14 Appendix B – References

This table lists the specifications for each elliptic curve in section 2.3

Curve	Specification
brainpoolP160r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP192r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP192t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP224r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP224t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP256r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP256t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP320r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP320t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP384r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP384t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP512r1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
brainpoolP512t1	http://www.ecc-brainpool.org/download/Domain-parameters.pdf
ec192wapi	http://www.gbstandards.org/GB_standards/GB_standard.asp?id=900 (The GB standard is available here for purchase)
nistP192	http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf
nistP224	http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf
numsP256t1	https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/curvegen.pdf
numsP384t1	https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/curvegen.pdf
numsP512t1	https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/curvegen.pdf
secP160k1	http://www.secg.org/sec2-v2.pdf
secP160r1	http://www.secg.org/sec2-v2.pdf
secP160r2	http://www.secg.org/sec2-v2.pdf
secP192k1	http://www.secg.org/sec2-v2.pdf
secP192r1	http://www.secg.org/sec2-v2.pdf
secP224k1	http://www.secg.org/sec2-v2.pdf
secP224r1	http://www.secg.org/sec2-v2.pdf
secP256k1	http://www.secg.org/sec2-v2.pdf
secP256r1	http://www.secg.org/sec2-v2.pdf
secP384r1	http://www.secg.org/sec2-v2.pdf
secP521r1	http://www.secg.org/sec2-v2.pdf
wtls12	http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf
wtls7	http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf
wtls9	http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf

Curve	Specification
x962P192v1	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P192v2	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P192v3	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P239v1	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P239v2	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P239v3	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)
x962P256v1	https://global.ihs.com/doc_detail.cfm?&item_s_key=00325725&item_key_date=941231&input_doc_number=ANSI%20X9%2E62&input_doc_title= (The ANSI X9.62 standard is available here for purchase)