

Non-proprietary Security Policy for FIPS 140-2 Validation

Code Integrity (ci.dll) in
Microsoft Windows 10
Windows 10 Pro
Windows 10 Enterprise
Windows 10 Enterprise LTSC
Windows 10 Mobile
Windows Server 2016 Standard
Windows Server 2016 Datacenter
Windows Storage Server 2016

DOCUMENT INFORMATION

Version Number	1.4
Updated On	August 6, 2019

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.

© 2019 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
12 DEC 2016	1.0	Tim Myers	First release to validators
28 FEB 2018	1.1	Mike Grimm	Update for build 10.0.14393.1770 (3SUB)
21 MAY 2018	1.2	Iffat Qamar	Bounded Modules added
25 FEB 2019	1.3	Garrett Burk	Maintenance Updates
6 AUG 2019	1.4	Garrett Burk	Updates in response to comments

TABLE OF CONTENTS

<u>1</u>	<u>INTRODUCTION.....</u>	<u>7</u>
1.1	LIST OF CRYPTOGRAPHIC MODULE BINARY EXECUTABLES	8
1.2	VERSION INFO	8
1.3	BRIEF MODULE DESCRIPTION	8
1.4	VALIDATED PLATFORMS	8
1.5	CRYPTOGRAPHIC BOUNDARY.....	8
<u>2</u>	<u>SECURITY POLICY.....</u>	<u>8</u>
2.1	FIPS 140-2 APPROVED ALGORITHMS	12
2.2	NON-APPROVED ALGORITHMS	12
2.3	CRYPTOGRAPHIC BYPASS.....	12
2.4	FIPS 140-2 APPROVED ALGORITHMS FROM BOUNDED MODULES	12
2.5	MACHINE CONFIGURATIONS	12
<u>3</u>	<u>INTEGRITY CHAIN OF TRUST</u>	<u>13</u>
3.1	CONVENTIONAL BIOS AND UEFI WITHOUT SECURE BOOT ENABLED	13
3.2	UEFI WITH SECURE BOOT ENABLED	13
<u>4</u>	<u>PORTS AND INTERFACES.....</u>	<u>13</u>
4.1	CODE INTEGRITY EXPORT FUNCTIONS.....	13
4.1.1	CIINITIALIZE().....	14
4.1.2	CIVALIDATEIMAGEHEADER()	14
4.1.3	CIVALIDATEIMAGEDATA().....	15
4.1.4	CIQUERYINFORMATION()	15
4.1.5	CIQUERYIMAGESIGNATURE()	15
4.1.6	CIIMPORTROOTS().....	15
4.1.7	CIGETFILECACHE()	15
4.1.8	CISSETFILECACHE()	15
4.1.9	CIHASHMEMORYSHA256()	16
4.1.10	CIGETPEINFORMATION().....	16
4.1.11	CIVERIFYHASHINCATALOG()	16
4.1.12	CICHECKSIGNEDFILE()	16
4.1.13	CIFINDPAGEHASHESINCATALOG().....	16
4.1.14	CIFINDPAGEHASHESINSIGNEDFILE().....	16

Code Integrity

4.1.15	CIFREEPOLICYINFO()	16
4.1.16	CIVALIDATEFILEOBJECT()	16
4.2	CONTROL INPUT INTERFACE	16
4.3	STATUS OUTPUT INTERFACE	16
4.4	DATA INPUT INTERFACE	17
4.5	DATA OUTPUT INTERFACE	17
<u>5</u>	<u>SPECIFICATION OF ROLES</u>	<u>17</u>
5.1	MAINTENANCE ROLES	17
5.2	MULTIPLE CONCURRENT INTERACTIVE OPERATORS	17
<u>6</u>	<u>SERVICES</u>	<u>17</u>
6.1	VERIFICATION OF INTEGRITY SERVICE	19
6.2	SHOW STATUS SERVICES	19
6.3	SELF-TEST SERVICES	19
6.4	SERVICE INPUTS / OUTPUTS	20
<u>7</u>	<u>OPERATIONAL ENVIRONMENT</u>	<u>20</u>
<u>8</u>	<u>AUTHENTICATION</u>	<u>20</u>
<u>9</u>	<u>CRYPTOGRAPHIC KEY MANAGEMENT</u>	<u>20</u>
9.1	CRITICAL SECURITY PARAMETERS	20
9.2	ACCESS CONTROL POLICY	21
<u>10</u>	<u>SELF-TESTS</u>	<u>21</u>
10.1	POWER-ON SELF-TESTS	21
10.2	CONDITIONAL SELF-TESTS	21
<u>11</u>	<u>DESIGN ASSURANCE</u>	<u>21</u>
<u>12</u>	<u>MITIGATION OF OTHER ATTACKS</u>	<u>23</u>
<u>13</u>	<u>SECURITY LEVELS</u>	<u>23</u>

<u>14</u>	<u>ADDITIONAL DETAILS</u>	<u>24</u>
<u>15</u>	<u>APPENDIX A – HOW TO VERIFY WINDOWS VERSIONS AND DIGITAL SIGNATURES</u>	<u>25</u>
15.1	HOW TO VERIFY WINDOWS VERSIONS	25
15.2	HOW TO VERIFY WINDOWS DIGITAL SIGNATURES	25

1 Introduction

Code Integrity is a feature that verifies the integrity of several key Windows operating system binary image files as they are loaded into memory from the disk.

Code Integrity is not a general-purpose cryptographic module. It is validated under FIPS 140-2 because it implements cryptographic algorithms and provides the integrity checks for the Windows general-purpose cryptographic modules.

Throughout this document, Code Integrity may also be referred to as CI and CI.DLL.

The Operational Environments (OEs) are:

1. Windows 10 Enterprise Anniversary Update (x86) running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
2. Windows 10 Enterprise Anniversary Update (x64) running on a Microsoft Surface Pro 3 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
3. Windows 10 Enterprise Anniversary Update (x64) running on a Microsoft Surface Pro 4 – Intel Core i5 with AES-NI and PCLMULQDQ and SSSE 3
4. Windows 10 Enterprise Anniversary Update (x64) running on a Microsoft Surface Book – Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
5. Windows 10 Enterprise Anniversary Update (x64) running on a Dell Precision Tower 5810MT - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
6. Windows 10 Enterprise Anniversary Update (x64) running on a HP Compaq Pro 6305 - AMD A4 with AES-NI and PCLMULQDQ and SSSE 3
7. Windows 10 Pro Anniversary Update (x86) running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
8. Windows 10 Pro Anniversary Update (x64) running on a Microsoft Surface Pro 3 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
9. Windows 10 Pro Anniversary Update (x64) running on a Microsoft Surface Pro 4 - Intel Core i5 with AES-NI and PCLMULQDQ and SSSE 3
10. Windows 10 Pro Anniversary Update (x64) running on a Microsoft Surface Book - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
11. Windows 10 Pro Anniversary Update (x64) running on a Dell Precision Tower 5810MT - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
12. Windows 10 Anniversary Update (x64) [consumer] running on a Microsoft Surface 3 - Intel Atom x7 with AES-NI and PCLMULQDQ and SSSE 3
13. Windows 10 Anniversary Update (x86) [consumer] running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
14. Windows 10 Anniversary Update (x64) [consumer] running on a Dell XPS 8700 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
15. Windows 10 Enterprise LTSB Anniversary Update (x86) running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
16. Windows 10 Enterprise LTSB Anniversary Update (x64) running on a Dell Precision Tower 5810MT - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
17. Windows 10 Enterprise LTSB Anniversary Update (x64) running on a Dell XPS 8700 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
18. Windows Server 2016 Standard Edition running on a HP Compaq Pro 6305 - AMD A4 with AES-NI and PCLMULQDQ and SSSE 3

Code Integrity

19. Windows Server 2016 Standard Edition running on a Dell PowerEdge R630 Server - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
20. Windows Server 2016 Datacenter Edition running on a Dell PowerEdge R630 Server - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
21. Windows Storage Server 2016 running on a Dell PowerEdge R630 Server - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
22. Windows 10 Mobile Anniversary Update running on a Microsoft Lumia 950 - Qualcomm Snapdragon 808 (A57, A53)

herein referred to as Windows 10 OEs.

All the computers for Windows 10 and Windows Server listed above are 64-bit Intel architecture and implement the AES-NI instruction set. The exceptions are:

- Dell Inspiron 660s - Intel Core i3

Windows 10 Mobile runs on the ARM architecture, which does not implement AES-Ni instructions:

- Microsoft Lumia 950 - Qualcomm Snapdragon 808 (A57, A53)

1.1 List of Cryptographic Module Binary Executables

CI.DLL – Version 10.0.14393.1770 for Windows 10 OEs

1.2 Version Info

10.0.14393.1770 for Windows 10 OEs

1.3 Brief Module Description

Code Integrity is a dynamically-linked library used to verify the integrity of other binary executable code files.

1.4 Validated Platforms

The Code Integrity component listed in Section 1.1 was validated using the machine configurations specified in the list of Windows 10 OEs.

1.5 Cryptographic Boundary

The cryptographic boundary for Code Integrity is defined as the enclosure of the computer system, on which Code Integrity is to be executed. The physical configuration of Code Integrity, as defined in FIPS 140-2, is multi-chip standalone.

2 Security Policy

Code Integrity is in a FIPS mode of operation when the following rules are followed:

- Code Integrity is supported on Windows 10 OEs.
- Windows 10 OEs are operating systems supporting a “single user” mode where there is only one interactive user during a logon session.

Code Integrity

- Code Integrity is only in the “Approved mode of operation” when Windows is booted normally, meaning Debug mode has not been enabled and Driver Signing enforcement has not been disabled.
- The Debug mode status and Driver Signing enforcement status can be viewed by using the bcdedit tool.
- Code Integrity operates in FIPS mode of operation only when used with the FIPS approved version of Windows 10 OEs Windows OS Loader (winload.exe or winload.efi) validated to FIPS 140-2 under Cert. #3502 operating in FIPS mode or Windows Resume (winresume.exe or winresume.efi) validated to FIPS 140-2 under Cert. #3501 operating in FIPS mode.

The following diagram, Figure 1, illustrates the master components of the Code Integrity module:

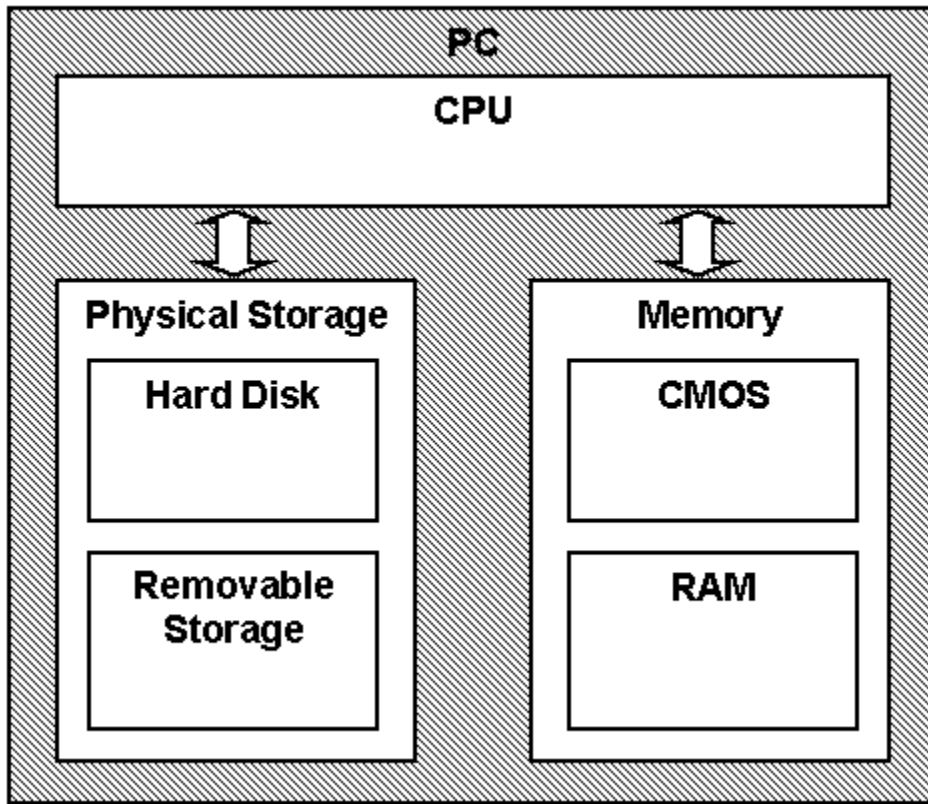


Figure 1

Code Integrity

The following diagram, Figure 2, illustrates the interaction of Code Integrity with other cryptographic modules:

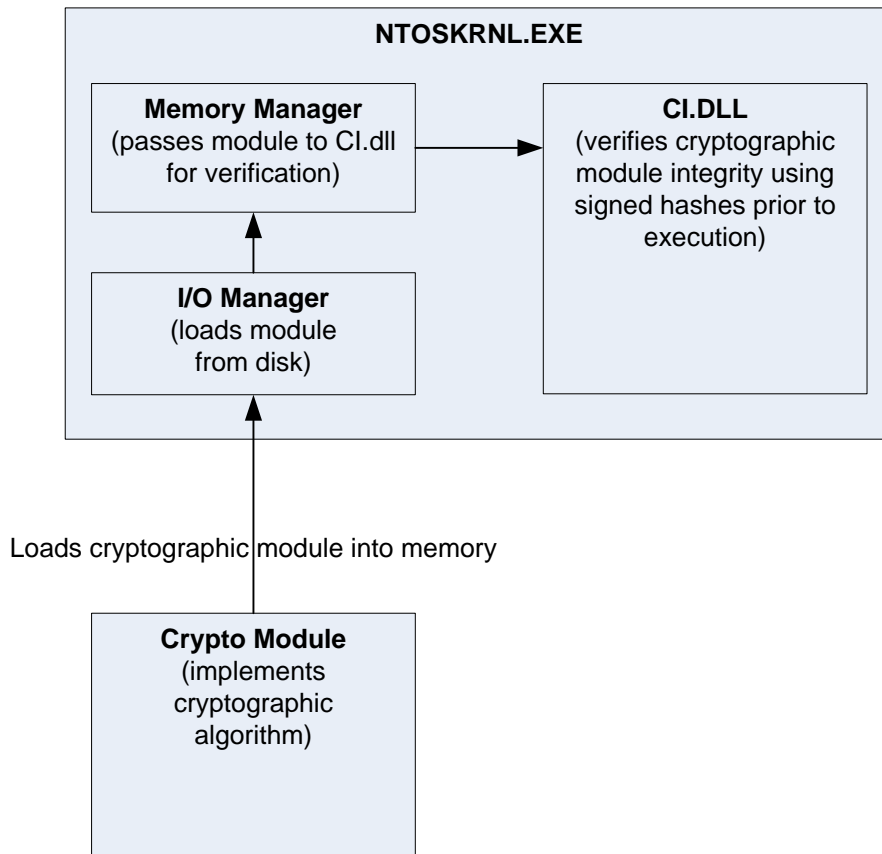


Figure 2

- Code Integrity's services are specified in Section 6 - Services.
- All services implemented within Code Integrity are available to the User and Crypto officer roles. The User and Crypto officer roles are assumed by the operating system or application processes that will invoke binary image verification in CI.dll. The Windows Memory Manager is an example.
- Code Integrity verifies the integrity of the Windows 10 OEs general purpose cryptographic modules using the following FIPS 140-2 Approved algorithms.
 - RSA PKCS#1 (v1.5) verify with public key
 - SHA-1 hash
 - SHA-256 hash
 - SHA-384 hash
 - SHA-512 hash

2.1 FIPS 140-2 Approved Algorithms

Code Integrity implements the following FIPS 140-2 Approved algorithms:

- FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 1024, 2048, and 3072 moduli; supporting SHA-1, SHA-256, SHA-384, and SHA-512 (Cert. # 2833)
- FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512 (Cert. # 4250)

Note that not all the algorithms and modes verified through the CAVP certificates listed are implemented by this module.

2.2 Non-Approved Algorithms

Code Integrity also includes a legacy implementation of MD5, which is not allowed for usage in FIPS mode. MD5 is only used for backwards compatibility to verify the RSA signature over the file digest and certificate chains. MD5 is not allowed for use in file digests, which require a SHA-1 hash as the minimum.

An instance of AES-ECB 128-bits is used in Code Integrity that does not undergo a self-test. It is not provided as a general-purpose encryption service for the protection of sensitive data at rest. It is not accessible by the User and Cryptographic Officer roles. See the `CiGetPEInformation()` function description below. It is not used in FIPS mode to provide any security to the module and any data that it is encrypting is considered plaintext.

2.3 Cryptographic Bypass

Cryptographic bypass is not supported by Code Integrity.

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, the Code Integrity depends on the following algorithms:

Implemented in Windows OS Loader (module #3502):

- CAVP certificate #2833 for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 supporting SHA-256
- CAVP certificate #4250 for FIPS 180-4 SHS SHA-256

Implemented in Windows Resume (module #3501):

- CAVP certificate #5295 for FIPS 197 AES CBC 128 and 256, SP 800-38E AES XTS¹ 128 and 256

2.5 Machine Configurations

Code Integrity was tested using the machine configurations listed in Section 1.4 - Validated Platforms.

¹ For XTS-AES the length of the data unit does not exceed 2^{20} blocks. XTS-AES mode is only used by the module for the cryptographic protection of data on storage devices.

3 Integrity Chain of Trust

3.1 Conventional BIOS and UEFI without Secure Boot Enabled

Boot Manager is the start of the chain of trust. It cryptographically checks its own integrity during its startup. It then cryptographically checks the integrity of the Windows OS Loader (Winload.exe) before starting it. The Windows OS Loader checks the integrity of Code Integrity, which is protected by an RSA signature with a 2048-bit key and SHA-256 message digest, before loading it into memory. Code Integrity is used to verify the origin and integrity of Windows system binaries before they are loaded into memory and executed. Code Integrity also ensures kernel mode drivers are appropriately signed. When User Mode Code Integrity (UMCI) is enabled, Code Integrity ensures that all binaries are appropriately signed.

3.2 UEFI with Secure Boot Enabled

On UEFI systems with Secure Boot enabled, Boot Manager is still the OS binary from which the integrity of all other OS binaries is rooted, and it does cryptographically check its own integrity. However, Boot Manager's integrity is also checked and verified by the UEFI firmware, which is the root of trust on Secure Boot enabled systems.

4 Ports and Interfaces

4.1 Code Integrity export functions

The following list contains all the functions exported by Code Integrity to its callers inside the kernel. Code Integrity is not callable outside the kernel. The exported functions are explained further in the subsequent subsections.

- CiInitialize()
- CiGetPEInformation()
- CiVerifyHashInCatalog()
- CiCheckSignedFile()
- CiFindPageHashesInCatalog()
- CiFindPageHashesInSignedFile()
- CiFreePolicyInfo()
- CiValidateFileObject()

The following functions are not exported, but are accessed via a callback structure provided by the CiInitialize() function. These functions are also explained in subsequent subsections.

- CiValidateImageHeader()
- CiValidateImageData()
- CiQueryInformation()
- CiQueryImageSignature()
- CiImportRoots()
- CiGetFileCache()
- CiSetFileCache()
- CiHashMemorySha256()

4.1.1 CiInitialize()

CiInitialize() is the function exported by Code Integrity for initializing the image file integrity validation capability of Code Integrity.

As the power-on (startup) function of Code Integrity, CiInitialize() conducts the following power-on (startup) self-tests.

- SHS (SHA-1) Known Answer Test
- SHS (SHA-256) Known Answer Test
- SHS (SHA-512) Known Answer Test
- RSA verify using a verify test with a Known Signatures of the PKCS#1 v1.5 format:
 - RSA signature with 1024-bit key and SHA-1 message digest
 - RSA signature with 2048-bit key and SHA-256 message digest

If a self-test fails, CiInitialize() returns STATUS_INVALID_IMAGE_HASH. On the other hand, after the successful initialization, CiInitialize() returns a callback structure consisting of the following functions. A caller subsequently can use these functions to obtain the image file integrity validation service from Code Integrity.

- CiValidateImageHeader()
- CiValidateImageData()
- CiQueryInformation()
- CiQueryImageSignature()
- CiImportRoots()
- CiGetFileCache()
- CiSetFileCache()
- CiHashMemorySha256()

4.1.2 CiValidateImageHeader()

When a caller (such as the Memory Manager) wants to obtain the set of trusted per-page hashes of an image file, it calls CiValidateImageHeader(). Trusted per-page hashes can use the following algorithms:

- SHS (SHA-1)
- SHS (SHA-256)
- SHS (SHA-384)
- SHS (SHA-512)

In the case of a Windows 10 OEs general purpose cryptographic module (namely, bcryptprimitives.dll, rsaenh.dll, or dssenh.dll), if CiValidateImageHeader() does not find the set of trusted per-page hashes for the cryptographic module, then CiValidateImageHeader() verifies the full cryptographic module image by verifying a trusted file hash. The trusted file hash may be:

- SHS (SHA-1)
- SHS (SHA-256)

Code Integrity

- SHS (SHA-384)
- SHS (SHA-512)

If this validation process fails, the cryptographic module is not valid. Subsequently, the Windows 10 OEs Memory Manager does not load any page of the cryptographic module.

Both the trusted file image hash and trusted page hashes are signed using the RSA signature algorithm with PKCS#1 v1.5 padding.

Code Integrity has a different verification procedure for kernel mode crypto modules that are loaded into memory all at once (not in a per-page fashion as the other user mode general purpose crypto modules). As a result, when `CiValidateImageHeader()` is called by the memory manager, the `CI_VALIDATE_DRIVER_IMAGE` flag is set, and the entire image is validated by verifying a trusted image hash. This is like the user mode module verification when page hashes are not present.

4.1.3 `CiValidateImageData()`

After calling `CiValidateImageHeader()` to obtain the set of trusted per-page hashes of an image file, a caller (such as the Memory Manager) would want to know the integrity of a page of the image file. The caller uses `CiValidateImageData()` to check the page integrity.

If the computed hash matches the identified trusted hash, then `CiValidateImageData` confirms the integrity of the page. Otherwise, `CiValidateImageData()` returns `STATUS_INVALID_IMAGE_HASH`. The Windows 10 OEs Enterprise Memory Manager does not load invalid pages.

4.1.4 `CiQueryInformation()`

Returns state data about the enforcement of Code Integrity. Whether CI is being enforced and whether test signing is enabled.

4.1.5 `CiQueryImageSignature()`

Returns whether a previously validated file is Windows signed (signing certificate chains to Microsoft Root and the Windows ECU). This check was done during a previous validation, and this function is just returning a cached result.

4.1.6 `CiImportRoots()`

Imports public keys that are used as trusted CAs for validation of user mode components.

4.1.7 `CiGetFileCache()`

For an input file, returns the previously validated signature level (MSFT, Windows, Authenticode) and the thumbprint of the signing certificate. This check was done during a previous validation, and this function is just returning a cached result.

4.1.8 `CiSetFileCache()`

For a verified file, saves the signature level and thumbprint of the signing certificate. If the file was not previously verified, it will verify the file against either its embedded signature or a system catalog.

4.1.9 CiHashMemorySha256()

Passes supplied data to CI's SHA256 implementation and returns the SHA256 hash of that data.

4.1.10 CiGetPEInformation()

Creates an AES-ECB encrypted channel between the caller and CI. It is used as part of protected media path DRM, and allows information about kernel drivers and user mode binaries loaded into protected processes to be returned to the caller. However, this AES implementation does not implement self-tests. For this reason, it is not considered to be providing any security to the module and any data that it is encrypting is considered plaintext.

4.1.11 CiVerifyHashInCatalog()

For an input Authenticode file digest, validates that the digest is contained within a verified system catalog. It optionally returns information about the catalog.

4.1.12 CiCheckSignedFile()

For an input Authenticode file digest and an Authenticode signature, verifies that the digest is in the signature and that the signature validates. It optionally returns information about the signature.

4.1.13 CiFindPageHashesInCatalog()

For an input Authenticode digest of the first page of a PE image, validates that the digest is contained within a verified system catalog. It optionally returns information about the catalog.

4.1.14 CiFindPageHashesInSignedFile()

For an input Authenticode digest of the first page of a PE image and an Authenticode signature, verifies that the digest is in the signature and that the signature validates. It optionally returns information about the signature.

4.1.15 CiFreePolicyInfo()

Frees memory allocated by the CiVerifyHashInCatalog(), CiCheckSignedFile(), CiFindPageHashesInCatalog(), and CiFindPageHashesInSignedFile() functions.

4.1.16 CiValidateFileObject()

Verifies the signature of a file object and returns the policy info along with the timestamp and signing time.

4.2 Control Input Interface

The Control Input Interface for Code Integrity consists of the three CI export functions. Options for control operations are passed as input parameters to the CI export functions. The SecureRequired parameter in CiValidateImageHeader() is the only control option provided by Code Integrity in the Control Input Interface.

4.3 Status Output Interface

The Status Output Interface for Code Integrity also consists of the three CI export functions. For each function, the status information is returned to the caller as the return value (e.g. STATUS_SUCCESS, STATUS_UNSUCCESSFUL, STATUS_INVALID_IMAGE_HASH) from the function.

4.4 Data Input Interface

The Data Input Interface for Code Integrity also consists of the three CI export functions. Data and options are passed to the interface as input parameters to the CI export functions. Data Input is kept separate from Control Input by passing Data Input in separate parameters from Control Input.

4.5 Data Output Interface

The Data Output Interface for Code Integrity also consists of the three CI export functions. For `CiInitialize()`, data is returned to its caller as the `Callbacks` output parameter. For `CiValidateImageHeader()`, data is returned to its caller as the `SePool` output parameter.

5 Specification of Roles

Code Integrity supports both User and Cryptographic Officer roles (as defined in FIPS 140-2). Both roles have access to all services implemented in Code Integrity through a caller component running in the kernel mode. The module does not provide authentication, as such both roles are implicitly assumed when the services exported by the module are invoked.

5.1 Maintenance Roles

Maintenance roles are not supported.

5.2 Multiple Concurrent Interactive Operators

There is only one interactive operator during a logon session. Multiple concurrent interactive operators sharing a logon session are not supported.

6 Services

Code Integrity's services are:

1. Verify the integrity of binary executable code.
2. Provide Show Status services that indicate whether the integrity checks passed.
3. Provide Self-Test services.
4. Legacy certificate chain authentication (non-FIPS Approved service)

Code Integrity does not offer any other services, operations, or functions that can be externally invoked. Code Integrity export functions are only available inside the kernel. The User and Cryptographic Officer roles are not able to invoke them directly.

The following table maps the services to their corresponding algorithms and critical security parameters (CSPs). All the Code Integrity export functions in section 4.1 map to the service to verify the integrity of binary executable code.

Table 1

Service / Function	Algorithms	CSPs	Invocation
Verify the integrity of binary executable code	FIPS 186-4 RSA PKCS#1 (v1.5) verify with public key FIPS 180-4 SHS: SHA-1 hash	Asymmetric Public keys	This service is fully automatic. The User / Cryptographic Officer does not take any actions to explicitly start this service. This

Code Integrity

	SHA-256 hash SHA-384 hash SHA-512 hash		service is executed whenever a binary executable is loaded.
Provide Show Status services that indicate whether the integrity checks passed	None	None	This service is fully automatic. The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed upon completion of an integrity check function.
Provide Self-Test services	FIPS 186-4 RSA PKCS#1 (v1.5) verify with public key and known signature FIPS 180-4 SHS: SHA-1 KAT SHA-256 KAT SHA-512 KAT	None	This service is fully automatic. The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed upon startup of this module.
Legacy certificate chain authentication (non-FIPS approved service)	MD5 (non-FIPS approved algorithm)	None	This service is fully automatic. The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever a binary executable with a legacy MD5 certificate is loaded.
CiGetPEInformation (non-FIPS approved function)	AES (non-FIPS approved algorithm)	None	This function is called as part of protected media path DRM operations. The User / Cryptographic Officer does not take any actions to explicitly invoke it.

The following table maps services and the export functions from Section 4 Ports and Interfaces.

Table 2

Service	Export Functions
Verify the integrity of binary executable code	CiInitialize() CiValidateImageHeader() CiValidateImageData() CiQueryImageSignature() CiImportRoots() CiGetFileCache() CiSetFileCache() CiHashMemorySha256() CiVerifyHashInCatalog() CiCheckSignedFile() CiFindPageHashesInCatalog()

	CiFindPageHashesInSignedFile() CiFreePolicyInfo() CiValidateFileObject()
Provide Show Status services that indicate whether the integrity checks passed	CiInitialize() CiValidateImageHeader() CiValidateImageData() CiQueryImageSignature() CiImportRoots() CiGetFileCache() CiSetFileCache() CiHashMemorySha256() CiVerifyHashInCatalog() CiCheckSignedFile() CiFindPageHashesInCatalog() CiFindPageHashesInSignedFile() CiFreePolicyInfo() CiValidateFileObject()
Provide Self-Test services	CiInitialize()
Legacy certificate chain authentication (non-FIPS approved service)	CiInitialize() CiValidateImageHeader() CiValidateImageData() CiQueryImageSignature() CiImportRoots() CiGetFileCache() CiSetFileCache() CiVerifyHashInCatalog() CiCheckSignedFile() CiFindPageHashesInCatalog() CiFindPageHashesInSignedFile() CiFreePolicyInfo() CiValidateFileObject()

There is also an export function that does not map to any service: CiQueryInformation().

6.1 Verification of Integrity Service

Code Integrity verifies the integrity of digitally signed drivers, Dynamic-linked Libraries (DLLs), and components within the computer (such as bcryptprimitives.dll, ncryptsslp.dll, and other binary executables).

6.2 Show Status Services

The status information is returned to the caller as the return value from the function. The User / Cryptographic Officer does not have any direct access to the return value, but rather, they may observe the failure of applications or services to load.

6.3 Self-Test Services

Code Integrity automatically executes Self-Tests upon being loaded, which provides the User / Cryptographic Officer assurance that the module is operating properly. Upon failing a Self-Test, Code

Code Integrity

Integrity will fail to load and return an error indicator (as described in section 4.1.1) which may be observed by the User / Cryptographic Officer as a failure of applications or services to load. The Self-Test functionality is described in Section 10 Self-Tests.

6.4 Service Inputs / Outputs

The User / Cryptographic Officer does not have access to the service inputs and outputs that are specified in Section 4 Ports and Interfaces.

7 Operational Environment

The operational environment for Code Integrity is the Windows 10 OEs running on the software and hardware configurations listed in Section 1.4 - Validated Platforms.

8 Authentication

Code Integrity does not implement any authentication services. The User and Cryptographic Officer roles are assumed implicitly by booting the Windows operating system. There are Code Integrity libraries that run before boot in Winload.exe and Winresume.exe. The CI.DLL is loaded in the kernel as part of the memory management path.

9 Cryptographic Key Management

Code Integrity does not handle security-relevant information such as secret and private cryptographic key, authentication data, nor any other protected information. Hence, there is no operation related to any of the below.

- Key generation
- Key output
- Key storage
- Key zeroization

The only cryptographic keys the module supports are the RSA PKCS#1 public keys used to verify integrity. These public keys are accessible by both approved roles. Due to such simplicity, an access control policy table is not included in this document. The public keys are stored on the hard-drive in plaintext.

9.1 Critical Security Parameters

The Code Integrity crypto module uses the following cryptographic keys, which are Critical Security Parameters (CSPs):

Table 3

Cryptographic Key	Key Description
-------------------	-----------------

Code Integrity

Asymmetric Public keys	Keys used for RSA PKCS#1 (v1.5) verification
------------------------	--

9.2 Access Control Policy

The Code Integrity crypto module does not contain CSPs that would require access controls.

10 Self-Tests

10.1 Power-On Self-Tests

Code Integrity performs the following power-on (startup) self-tests:

- SHS (SHA-1) Known Answer Test
- SHS (SHA-256) Known Answer Test
- SHS (SHA-512) Known Answer Test
- RSA verify using a verify test with a Known Signature of the PKCS#1 v1.5 format with both 1024-bit keys with SHA1 digest and 2048-bit keys with SHA-256 digest.

The integrity of Code Integrity itself is protected by an RSA signature with a 2048-bit key and SHA-256 message digest, which is verified by Winload.exe before Code Integrity is loaded into memory. If the self-test fails, the module will not load and status will be returned. If the status is not STATUS_SUCCESS, then that is the indicator a self-test failed.

10.2 Conditional Self-Tests

Code Integrity does not perform conditional self-tests.

11 Design Assurance

The secure installation, generation, and startup procedures of this cryptographic module are part of the overall operating system secure installation, configuration, and startup procedures for the Windows 10 OEs. The various methods of delivery and installation for each product are listed in Table 4.

Table 4

Product	Delivery and Installation Method
Windows 10, Windows 10 Pro, Windows 10 Enterprise, Windows Enterprise LTSC, Windows Server 2016 Standard, Windows Server 2016 Datacenter	<ul style="list-style-type: none">• Pre-installed on the computer by OEM• Download that updates to Windows 10• Enterprise IT deployment

Code Integrity

Surface Book, Surface Pro 4, Surface Pro 3, Surface 3, Lumia 950, Windows Storage Server 2016	<ul style="list-style-type: none">• Pre-installed by the OEM (Microsoft)
---	--

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

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

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

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

12 Mitigation of Other Attacks

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

Table 5

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
	Cache Attack	Memory Access pattern is independent of any confidential data

13 Security Levels

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

Table 6

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

14 Additional Details

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

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

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

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

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

15.1 How to Verify Windows Versions

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

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

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

15.2 How to Verify Windows Digital Signatures

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

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