Federal Office
for Information Security

Work Package 7: Device Guard

Version: 1.0

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-0
E-Mail: bsi@bsi.bund.de
Internet: https://www.bsi.bund.de

Table of Contents

Figures

Tables

1 Introduction

1.1 Zusammenfassung

Dieses Kapitel stellt das Ergebnis von Arbeitspaket 7 des Projekts „SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ dar. Das Projekt wird durch die Firma ERNW GmbH im Auftrag des Bundesamts für Sicherheit in der Informationstechnik (BSI) durchgeführt.

Ziel dieses Arbeitspakets ist die Analyse des Device Guard Features des Microsoft Windows 10 Betriebssystems. Wie durch das BSI vorgegeben wird Windows 10 Build 1607, 64-bit, long-term servicing branch (LTSB), Deutsch betrachtet.

Die beschriebenen Analysen wurden dynamisch und statisch durchgeführt; jeweils unter Nutzung von `windbg` oder entsprechend `IDA` Dissassembler. Die technischen Beschreibungen umfassen Call Stacks von Funktionen und Pseudo Code. Die Call Stacks enthalten der Übersichtlichkeit wegen nur Funktionen, die für das Arbeitspaket relevant sind. Der Pseudo Code ist vom echten Code abstrahiert und nicht in einer bestimmten Programmiersprache definiert, basiert aber auf C/C++. Die Analyse der Funktionsteile, die in einer sicheren Umgebung ausgeführt werden, wurde mit rein statischen Methoden durchgeführt. Sichere Umgebungen sind in [ERNW WP6] beschrieben und sind aktiv, wenn der Virtual Secure Mode (VSM) und Hypervisor Code Integrity (HVCI) aktiviert sind. SecureBoot ist eine Voraussetzung für HVCI und macht die Nutzung eines Debuggers für die dynamische Analyse sicherer Umgebungen unmöglich.

Die folgenden Abschnitte fassen die erarbeiteten Ergebnisse zusammen und verweisen auf die jeweiligen Kapitel für die ausführlichere Beschreibung.

Architekturüberblick (Kapitel 1.3) Die Device Guard Komponente von Windows 10 implementiert ein Feature zur Verhinderung der Ausführung nicht-vertrauenswürdigen Codes. „Nicht-vertrauenswürdiger Code“ sind Programme, deren Integrität und/oder Authentizität nicht verifiziert werden können (weil diese beispielsweise verändert oder aus nicht bekannten Quellen heruntergeladen wurden). Device Guard implementiert das Feature „konfigurierbare Codeintegrität“, das Nutzer-definierte Richtlinien für die Verifikation von Images (d.h. ausführbare Dateien) umsetzt und erzwingt [ms_dg]. Die Richtlinien können kryptografische Informationen (wie beispielsweise Hash Werte) oder nicht-kryptografische Informationen (wie Dateinamen) nutzen. Zusätzlich zu den konfigurierbaren Richtlinien verwendet Windows 10 einen Codeintegritäts-Feature, welcher nicht durch den Nutzer konfiguriert werden kann (vergleiche auch [ERNW WP5], Kapitel 2.2). Dieses Feature wird im vorliegenden Dokument als nicht-konfigurierbare Codeintegrität bezeichnet.

Das konfigurierbare Codeintegritäts-Feature kann wiederum in zwei Kategorien unterteilt werden: User-Mode Codeintegrität (UMCI) und Kernel-Mode Codeintegrität (KMCI) ([Yosif 2017], Kapitel 7). UMCI deckt Entitäten ab, die im Userland ausgeführt werden, KMCI Kernelland Entitäten (wie den Kernel selbst sowie dessen Erweiterungen und Treiber). Die UMCI und KMCI Implementierungen der konfigurierbaren Codeintegrität sind ebenfalls als Windows Defender Application Control (WDAC) bekannt (siehe Abbildung 1).

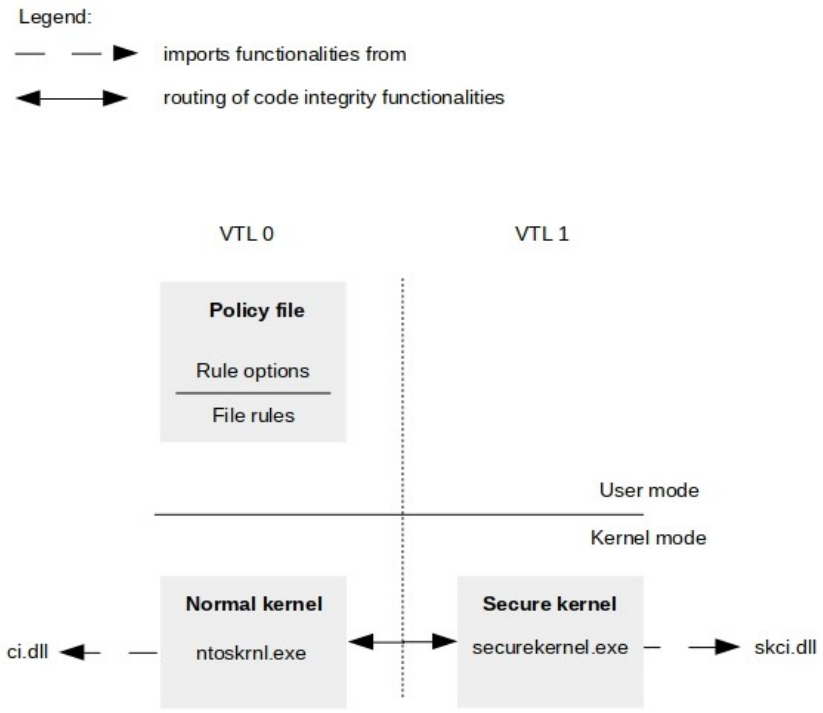

Abbildung 1: Die Architektur der Codeintegrität-Features von Device Guard und Windows 10 (Überblick)

Beide Codeintegritäts-Features implementieren Funktionalität im Boot Manager, im Windows Loader und im Windows Kernel. Boot Manager und Windows Loader enthalten die Funktionalität direkt in den ausführbaren Dateien, der Kernel bindet externe Routinen aus Bibliotheken ein. Falls das VSM Feature HVCI deaktiviert ist, wird Codeintegritäts-Funktionalität im Kontext der `ci.dll` Bibliothek ausgeführt. Diese Bibliothek wird durch `ntoskrnl.exe` geladen (welche den Normal Kernel darstellt, vergleiche [ERNW WP6], Kapitel 1.3). Falls HVCI aktiv ist, leitet Windows Codeintegritäts-Funktionalität zur Ausführung an eine sichere Umgebung weiter (VTL 1), wo die Funktionalität dann in der `skci.dll` Bibliothek ausgeführt wird. Diese Weiterreichung verhindert die Modifikation von Codeintegritäts-Funktionalität durch einen Angreifer mit Zugang zur normalen Umgebung. `skci.dll` wird durch den `securekernel.exe` geladen, der den Sicheren Kernel implementiert ([ERNW WP6], Kapitel 1.3).

WDAC verifiziert Images basierend auf Nutzer-definierten Richtlinien, die unter anderem Dateinamen, -versionen und -hashes referenzieren können. Die Verifikation basiert aus dem Vergleich der in der Richtlinie hinterlegten Daten mit den tatsächlichen Image-Daten. Beispielsweise kann eine Regel den Hash-Wert eines Images beinhalten. Bei der Verifikation wird dann der Hash-Wert aus der Regel mit dem berechneten Hashwert des auszuführenden Images verglichen. Falls die Werte nicht übereinstimmen, wird das Image nicht ausgeführt. Nutzer-Richtlinien werden in einer sogenannten WDAC Richtlinie gespeichert. Diese Datei wird im Extensible Markup Language (XML) Format erstellt und dann für die Installation in ein Binärformat umgewandelt. Die Binärdatei kann digital signiert werden, um Veränderungen zur Laufzeit zu verhindern.

Eine WDAC Richtlinie kann die folgenden Elemente enthalten:

- Globale Richtlinien („policy rule options“): Globale Richtlinien konfigurieren das allgemeine Verhalten von WDAC. Beispielsweise aktiviert `Enabled: UMCI` die Userland Codeintegrität;
- Dateiregeln (file rules): Dateiregeln definieren die Verifikation von Images basierend auf einem zugewiesenen Level. Diese Level (im vorliegenden Dokument Policy Level genannt) definieren, auf welche Art ein Image verifiziert wird. Auf Level `Hash` wird das Image beispielsweise basierend auf seinem Hashwert verifiziert.

Ein Nutzer kann eine WDAC-Richtlinie automatisiert über das PowerShell cmdlet `New-CIPolicy` erzeugen, wobei Datei-Regeln für Images unter einem bestimmten Pfad erzeugt werden. Diese Regeln ermöglichen das Verbot oder die Freigabe des jeweiligen Images basierend auf der Einschätzung des Nutzers über das Image. Windows 10 speichert WDAC Richtlinien in der Datei `SIPolicy.p7b`. Auf Systemen ohne UEFI (Unified Extensible Firmware Interface) Unterstützung wird die Datei `SIPolicy.p7b` unter dem Pfad `%System%\System32\CodeIntegrity\` gespeichert. Auf Systemen mit UEFI-Unterstützung speichert Windows 10 `SIPolicy.p7b` zusätzlich in dem Verzeichnis `\EFI\Microsoft\Boot\` der Boot-Partition ([ERNW WP5], Kapitel 2.2.1).

Nicht-konfigurierbare Codeintegrität: Funktionsüberblick (Kapitel 2.1) Die nicht-konfigurierbare Codeintegrität verifiziert die Integrität von Code basierend auf der Authenticode Technologie ([ERNW WP5], Kapitel 2.2.3). Die Bibliotheken `ci.dll` und `skci.dll` werden durch den Normalen bzw. sicheren Kernel geladen und exportieren die entsprechende Codeintegritäts-Funktionalität.

`ci.dll` exportiert 26 Funktionen an den Normalen Kernel; `skci.dll` exportiert 9 Funktionen an den sicheren Kernel. Diese Funktionen bieten unterschiedliche Funktionalität an und können wie folgt klassifiziert werden:

- *Initialisierung*: Verschiedene Funktionen werden für die Initialisierung der Codeintegrität benötigt. Dies umfasst Selbst-Tests inklusive der Integritätsprüfung der Codeintegrität selbst.
- *Integritätsprüfung*: Funktionen welche die eigentliche Integritätsprüfung von Images vornehmen.
- *Verschiedenes*: Unterstützende Funktionen wie bspw. Speicher-Allokation/-Freigabe, Informationsabfragen und kryptografische Operationen wie Hashing.

Normaler und sicherer Kernel verifizieren die Integrität von `ci.dll` bzw. `skci.dll` um diese vor unautorisierter Veränderung zu schützen. Die Funktionen aus `skci.dll` werden genutzt, wenn Windows 10 Codeintegritäts-Funktionalität an den sicheren Kernel weiterreicht; diese Weiterreichung findet über dediziert aufgebaute und verifizierte Kanäle statt. Diese Kanäle werden als Secure Services bezeichnet und sind Teil des VSM Features von Windows 10 ([ERNW WP6], Kapitel 2.2.3). Der Datenaustausch zwischen den Kerneln umfasst Marshalling und Eingabeüberprüfung um das Risiko von Implementierungsfehlern im sicheren Kernel zu reduzieren.

WDAC: Initialisierung (Kapitel 2.2.1) WDAC wird beim Windows 10 Boot durch Windows Loader und Kernel initialisiert ([ERNW WP5], Kapitel 2.2), was das Laden der installierten *WDAC*-Richtlinie umfasst. Dabei wird unter anderem die Integrität und die Signatur der Richtlinie verifiziert um die enthaltenen Regeln vor unautorisierter Veränderung zu schützen. Dementsprechend wird empfohlen, die *WDAC*-Richtlinie nur signiert zu installieren. Der Windows Loader verifiziert die Richtlinie gemäß (Public Key Cryptography Standards) *PKCS#7* Standard [rfc_pkcs]. Die Integrität des Windows Loaders selbst wird durch den Boot Manager verifiziert ([ERNW WP5], Kapitel 2.2), um so auch unautorisierte Veränderungen an den Lade-Funktionen für die *WDAC*-Richtlinie zu unterbinden. Der Windows Loader kann die Richtlinie nach dem Laden für Codeintegritäts-Prüfungen verwenden. Abschließend übergibt der Loader die Ausführungskontrolle an den Kernel, welcher `ci.dll` und *WDAC*-Richtlinie erneut lädt und für entsprechende Prüfungen verwendet.

WDAC: Verifikation (Kapitel 2.2.2) Die Verifikation von Images beginnt in der Funktion `SIPolicyValidateImage` aus `ci.dll` basierend auf den Regeln aus der installierten WDAC-Richtlinie. Um die Ausführung eines Images zu erlauben werden die Daten der Richtlinie mit Laufzeit-Daten verglichen: Beispielsweise wird der Hashwert eines Images mit dem Hashwert der in der Richtlinie gespeichert ist verglichen, um zu ermitteln, ob die Ausführung des Images erlaubt ist. Die Vergleiche werden mittels typischer sicherer Funktionen zum Vergleich von Speicher und Zeichenketten angestellt (wie zum Beispiel `memcmp` [ms_memcmp] und `RtlEqualUnicodeString` [ms_str]), die standardmäßig im Windows Kernel enthalten sind. Dementsprechend ist auch die Integrität der Vergleichsfunktionen (als kritisches Element der Codeintegrität) über die Integritätsprüfung des Kernels ([ERNW WP5], Kapitel 2.2) sichergestellt. Damit ist die Integrität aller Verifikationskomponenten sichergestellt; die Korrektheit des Inhalts der WDAC-Richtlinie sowie deren Signatur liegt allerdings im Verantwortungsbereich des Nutzers:

Die Integrität von Images basiert rein auf Daten über das Image, es findet keine semantische Prüfung (beispielsweise ob die Funktion des Images ausschließlich gutartig ist) statt.

Daten für die Imageverifikation werden direkt von den Images abgeleitet, die in der WDAC-Richtlinie hinterlegt sind. Diese Ableitung findet ebenfalls über Kernel-Funktionen statt. Die Analyse der Implementierung oder des Designs der Funktionalität für Vergleich und Erstellung der WDAC-Richtlinie ist nicht im Fokus dieses Dokuments.

Konfiguration und Logging (Kapitel 3.1 und Kapitel 3.2) WDAC-Richtlinien können über entsprechende PowerShell cmdlets erstellt und angepasst werden. Diese cmdlets sind Teil des Powershell Moduls `ConfigCI`. Der `Group Policy Object Editor` kann für die Installation einer WDAC-Richtlinie verwendet werden: Der Pfad `Computer Configuration\Administrative Templates\System\ Device Guard\Deploy Code Integrity Policy` erlaubt die Definition der zu installierenden Richtlinien-Datei.

Die Optionen in der Richtlinien-Datei erlauben flexible Konfiguration mit Hinblick auf Striktheit der Prüfung und Wartbarkeit der Richtlinie. Beispielsweise ermöglicht der Policy Level `Hash` die granulare Freigabe genauer Image-Inhalte (wie sie bspw. auf sehr sicherheitskritischen Systemen notwendig ist). Gleichzeit resultiert dies in einem hohen Pflegeaufwand, da die Richtlinie bei jeder Image-Änderung (wie bspw. einem Update) angepasst werden muss. Entsprechend muss die Konfiguration der Richtlinie auf die verfügbaren Betriebsressourcen und das gewünschte Verifikationslevel angepasst werden. Die folgenden Punkte müssen daher für die erfolgreiche Nutzung von WDAC beachtet werden:

- *Ziel-Definition der WDAC-Installation*: WDAC erlaubt die Ausführungskontrolle ausschließlich auf der Gesamt-Systemebene. Sollte eine Ausführungskontrolle auf Benutzer- oder Benutzergruppen-Ebene notwendig sein müssen alternative Mechanismen (wie beispielsweise Microsoft AppLocker [ms_appl]) hinzugezogen werden.
- *Identifikation von Limitierungen*: Nicht alle Systeme können effizient und effektiv durch WDAC geschützt werden. Dies umfasst beispielsweise Systeme zur Entwicklung von Code, da auf diesen häufig unbekannte Images ausgeführt werden müssen, was die Konfiguration einer WDAC-Richtlinie schwer bis kaum betreibbar macht;
- *Entfernung von Images die Code-Ausführung ermöglichen*: WDAC kann umgangen werden, wenn die Ausführung eines Images gestattet ist, das wiederum andere Images lädt (wie beispielsweise einen Debugger). Entsprechend muss ein Prozess existieren, um solche Images kontinuierlich zu identifizieren und von den Systemen zu entfernen und/oder diese in der WDAC-Richtlinie zu blocken;
- *Management von Zertifikaten und Signaturprozess*: Eine WDAC-Richtlinie sollte vor ihrer Installation digital signiert werden um unautorisierte Änderungen zu verhindern. Diese Signatur sollte auf einem entsprechend gehärteten System stattfinden und erfordert, dass die Verwaltung des Zertifikats und dessen Lebenszyklus ebenfalls auf eine sichere Art und Weise geregelt ist;
- *Analyse und Überwachung*: Durch WDAC erzeugte Ereignisse müssen aufgenommen und nachverfolgt werden, um Richtlinien-Verletzungen analysieren zu können und so potentielle Sicherheitsvorfälle identifizieren zu können. Dies erfordert den Aufbau entsprechender Aufzeichnungs- und Auswertungs-Prozesse;
- *Prozess für die Verwaltung des Image-Lebenszyklus:* Es sollte ein Prozess etabliert werden, der die Freigabe neuer Software und deren Images mit der Anpassung der WDAC-Richtlinie(n) verknüpft und so einen effizienten Schutz der Systeme ermöglicht.

Die Aufzeichnung von WDAC Ereignissen findet mittels des Event Tracing for Windows (ETW) Frameworks statt ([ERNW WP2], Section 4.1). Tabelle 1 enthält die Namen und Globally Unique Identifiers (GUIDs) der relevanten ETW Provider.

ETW Provider	GUID
Microsoft-Windows-Applocker	CBDA4DBF-8D5D-4F69-9578-BE14AA540D22
Microsoft-Windows-CodeIntegrity	4EE76BD8-3CF4-44A0-A0AC-3937643E37A3
Microsoft-Windows-DeviceGuard	F717D024-F5B4-4F03-9AB9-331B2DC38FFB

Tabelle 1: ETW Provider für Codeintegrität Ereignisse (Überblick)

Zusammenfassung Die konfigurierbare (WDAC) und nicht-konfigurierbare Codeintegritäts-Features verhindern erfolgreich die Ausführung nicht-vertrauenswürdigen Codes. Die WDAC-Richtlinien erlauben die flexible Verifizierung der Vertrauenswürdigkeit von Images basierend auf benutzerdefinierte Kriterien. Dies erfordert aufwendige betriebliche Prozesse um die WDAC-Richtlinien an kontinuierlich veränderte IT-Umgebungen anzupassen.

Die Implementierung der konfigurierbaren (WDAC) und nicht-konfigurierbaren Codeintegritäts-Features ist robust. Diese Features sind durchgängig durch Boot Manager, Windows Loader, Windows Kernel und Bibliotheken implementiert und deren Integrität in einer durchgängigen Kette sichergestellt. Die konfigurierbare Funktionalität der Device Guard Komponente (WDAC) bietet viele Konfigurationsoptionen, wie zum Beispiel `Enabled: Managed Installer` und `Enabled: Intelligent Security Graph Authorization`. Diese Konfigurationsoptionen delegieren den Verifikationsprozess von Images an externe Entitäten, wie beispielsweise `AppLocker` oder `Intelligent Security Graph` [ms_inteli]. Sie sind nicht Teil dieses Arbeitspakets und sollten noch detailliert untersucht werden um ihre Auswirkung auf die Effektivität von Device Guard bewerten zu können.

1.2 Executive Summary

This chapter implements the work plan outlined in Work Package 7 of the project “SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ (orig., ger.). The project is contracted by the German Federal Office for Information Security (orig., ger., Bundesamt für Sicherheit in der Informationstechnik - BSI). The work planned as part of Work Package 7 has been conducted by ERNW GmbH in the time period between June 2018 and October 2018, in accordance with the time plan agreed upon by ERNW GmbH and the German Federal Office for Information Security.

The objective of this work package is the analysis of the Device Guard feature of Windows 10. As required by the German Federal Office for Information Security, the release of the Windows 10 system in focus is build 1607, 64-bit, long-term servicing branch (LTSB), German language.

The analysis presented in this work was performed by applying static and dynamic code analysis methods using the `windbg` debugger and the `IDA` disassembler. Analysis of functionalities or implementations deployed in the secure environment was performed by applying static analysis methods only. These functionalities are active when virtual secure mode (VSM) and hypervisor code integrity (HVCI) are enabled ([ERNW WP6], Section 1.3.2). SecureBoot, which is a requirement for HVCI ([ERNW WP6], Section 3.1), makes the use of a debugger for the purpose of dynamic analysis an unfeasible task.

The following paragraphs provide a summarizing overview of relevant analysis results. The referenced sections provide more details on the discussed topics.

Architecture overview (Section 1.3) The Device Guard component of Windows 10 implements a feature for preventing the execution of untrusted code. Untrusted code is program code whose integrity and authenticity cannot be verified. For example, this is code that has been tampered with in an unauthorized manner, or originates from untrusted sources. Device Guard implements a feature referred to as configurable code integrity. Configurable code integrity takes user-defined criteria into account in order to verify images, that is, to allow only specific images – executable files – to execute [ms_dg]. These criteria may involve cryptographic information (e.g., hash values) or non-cryptographic information (e.g., file names). In addition to configurable code integrity, Windows 10 implements code integrity functionalities that do not take user-defined criteria into account ([ERNW WP5], Section 2.2). This work refers to these functionalities as non-configurable code integrity.

The configurable code integrity feature of Device Guard can be structured into two categories: user-mode code integrity (UMCI) and kernel-mode code integrity (KMCI) ([Yosif 2017], Chapter 7). UMCI is for entities that operate in user-mode. KMCI is for entities that operate in kernel-mode. This includes the kernel and its extensions, such as drivers. The UMCI and KMCI implementations of the configurable code integrity feature are also known as Windows Defender Application Control (WDAC).

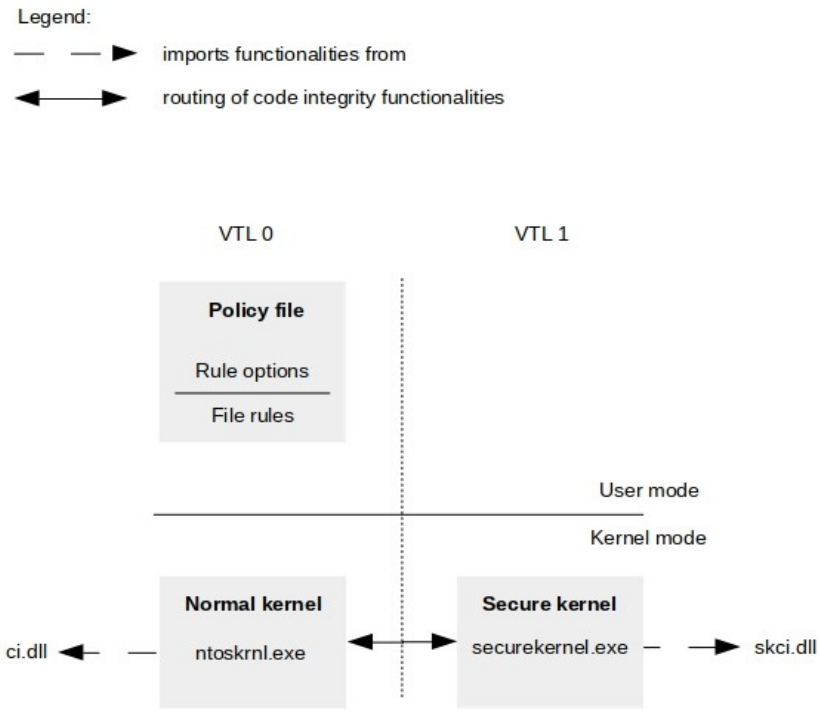

Figure 2: The architecture of the Device Guard and Windows 10 code integrity features (summarizing overview)

Figure 2 depicts an overview of the architecture of the Device Guard and Windows 10 code integrity features. The configurable and non-configurable code integrity features implement functionalities in the boot manager, the Windows loader, and the Windows kernel. In the context of the boot manager and the Windows loader, these functionalities are implemented as part of their executables. In the context of the Windows kernel, code integrity functionalities are implemented as kernel routines in external library files. If the VSM feature HVCI is disabled, code integrity functionalities are executed in the context of the `ci.dll` library file. This file is loaded by the `ntoskrnl.exe` executable, which implements the normal kernel ([ERNW WP6], Section 1.3). The `ci.dll` library file exposes an interface of functions to the kernel for use. If HVCI is enabled, Windows routes code integrity functionalities to the secure environment, that is, to the virtual trust level (VTL) 1, for execution. Code integrity functionalities are then executed in the context of the `skci.dll` library file. This prevents attackers that have gained access to the normal environment to tamper with critical code integrity functionalities. `skci.dll` is loaded by the `securekernel.exe` executable, which implements the secure kernel ([ERNW WP6], Section 1.3).

WDAC performs image verification based on user-defined rules. Among other things, these rules may specify file names, file versions, and hashes of images. An image is verified based on comparing rule-specified data with relevant data associated with the image. For example, a rule may specify the image's hash value. When the image is verified, Windows compares the rule-specified hash value with a hash value that it has calculated. In the case of a mismatch, the image may not be allowed to execute.

User-defined rules are stored in a policy file, referred to as WDAC policy. This file is written in the Extensible Markup Language (XML) format and then converted into binary format for deployment. The WDAC policy can be digitally signed in order to prevent unauthorized modifications after it is deployed.

In a WDAC policy, a user may define:

- policy rule options: Policy rule options configure the overall functionality of WDAC. An example is the `Enabled: UMCI` option, which enables UMCI;
- file rules: File rules configure verification for images. This configuration is done based on associating a specific level with file rules. Such levels specify at what level a given image is trusted. This work refers to

these levels as policy levels. An example policy level is `Hash`, which verifies an image based on its hash value.

A user may generate a WDAC policy automatically by using the `New-CIPolicy` PowerShell cmdlet [ms_newci]. This cmdlet allows for the automatic creation of WDAC policies such that it generates file rules for images stored at a user-specified path. These rules may allow or block the execution of these images based on the user's evaluation on the trustworthiness of the images.

Non-configurable code integrity: Overview of functionalities (Section 2.1) The non-configurable code integrity feature verifies code integrity based on the established Microsoft Authenticode digital signing technology ([ERNW WP5], Section 2.2.3). The library files `ci.dll` and `skci.dll` are loaded by the normal and secure kernel, respectively, and provide code integrity services to them in the form of exported functions.

`ci.dll` exports 26 functions to the normal kernel. `skci.dll` exports 9 functions to the secure kernel. The functions exported by `ci.dll` and `skci.dll` functions provide a variety of functionalities. These can be categorized as follows:

- *initialization:* To this category belong functionalities that are related to the initialization of code integrity functionalities. This includes the execution of self-tests for evaluating functional correctness and the integrity of the implementations of the code integrity functionalities themselves;
- *image integrity verification:* To this category belong functionalities that perform image integrity verification;
- *miscellaneous*: To this category belong functionalities that do not fit into any of the categories above. Such functionalities perform activities such as memory allocation and deallocation, status and information querying, and cryptographic operations, such as hashing.

The normal and the secure kernel verify the integrity of the `ci.dll` and `skci.dll` library files. This makes the code integrity functionalities implemented as part of them secure against unauthorized modifications.

The functions exported by `skci.dll` are invoked when Windows 10 routes code integrity functionalities from the normal to the secure kernel. This routing is conducted through established and security vetted channels for that purpose. These channels, referred to as secure services, are implemented as part of the VSM feature of Windows 10 ([ERNW WP6], Section 2.2.3). This implies that any data related to code integrity that is passed between the normal and the secure kernel is marshalled and sanitized. This is a security measure for checking, controlling, and managing this data. This significantly reduces the risk of exploiting implementation or design errors in implemented code integrity functionalities through the malicious manipulation of passed data related to code integrity.

WDAC: Initialization (Section 2.2.1) WDAC is initialized by the Windows loader and the kernel when Windows 10 is booted ([ERNW WP5], Section 2.2). This involves initializing and loading the deployed *WDAC* policy in the execution context of the Windows loader and the kernel. When loading a *WDAC* policy, among other things, the Windows loader verifies the integrity of the policy, if the policy is signed. This makes the file rules stored in the policy, based on which image verification is conducted, secure against unauthorized modifications. Therefore, it is strongly recommended for a *WDAC* policy to be signed before deployment. The Windows loader verifies the signature of a deployed WDAC policy with respect to the established (Public Key Cryptography Standards) *PKCS#7* standard [rfc_pkcs]. The integrity of the Windows loader itself is verified by the boot manager ([ERNW WP5], Section 2.2). This prevents unauthorized modifications of the functions for loading the *WDAC* policy, which are implemented as part of the Windows loader.

Once the Windows loader has loaded a *WDAC* policy, it may use the policy for image verification. When then Windows loader is finished executing, it transfers the execution control to the kernel. The kernel then loads the `ci.dll` library file and the *WDAC* policy. At this point, this policy may be used for image verification by the kernel.

WDAC: Verification (Section 2.2.2) Windows 10 performs *WDAC* verification in the `SIPolicyValidateImage` function. This function is implemented in `ci.dll`.

`SIPolicyValidateImage` verifies images based on data stored in a deployed WDAC policy and brings the decision whether an image is allowed to execute. `SIPolicyValidateImage` verifies an image based on comparing:

- data stored in a deployed WDAC policy as part of file rules; with
- verification data associated with the image being verified. This work refers to this data as image verification data.

What image verification data and data stored in WDAC policy is compared in `SIPolicyValidate`, depends on the policy levels configured in the policy. For example, this data includes the image's hash value if the policy level `Hash` is configured.

`SIPolicyValidateImage` compares data using standard and safe data comparison functions, such as memory and string comparison functions. Examples include `memcmp` [ms_memcmp] and `RtlEqualUnicodeString` [ms_str]. These functions are implemented as part of the Windows kernel. Therefore, their integrity is verified when the integrity of the kernel is verified. The Windows loader verifies the integrity of the kernel ([ERNW WP5], Section 2.2).

The data compared in `SIPolicyValidateImage` originates from trustworthy sources. Data stored in a deployed WDAC policy may be considered trustworthy if the policy is signed. This emphasizes the importance of signing a WDAC policy before deployment. The Windows loader verifies the signature of the WDAC policy. If this verification fails, the kernel does not perform image verification based on the policy.

It is important to note that the functional correctness of the file rules stored in a WDAC policy is a responsibility of the user creating the policy. For example, a user evaluates whether a given image is trustworthy. The user then generates a file rule for the image and stores it in the WDAC policy. This rule allows the execution of the image. However, the image may have been falsely evaluated as trustworthy by the user. In such a scenario, the WDAC policy will treat the image as trustworthy nevertheless. This is because of the file rule for the image specified in it.

Image verification data is directly extracted from the images that are associated with file rules stored in the deployed WDAC policy. The extraction is conducted by several functions implemented as part of the kernel. The integrity of these functions is verified when the integrity of the kernel is verified.

The analysis of implementation or design errors of the functionalities that compare and generate data stored in a WDAC policy and image verification data, is out of the scope of this work.

Configuration and logging capabilities (Section 3.1 and Section 3.2) WDAC policies can be created and modified by executing PowerShell cmdlets for WDAC management. These cmdlets are implemented as part of the PowerShell module `ConfigCI`. The `Group Policy Object Editor` utility can be used for deploying a WDAC policy. The group policy for deploying a WDAC policy is located at the policy path `Computer Configuration\Administrative Templates\System\Device Guard\Deploy Code Integrity Policy`.

The policy rule options and policy levels make WDAC highly configurable. They allow for security administrators to decide on a trade-off between policy manageability and verification strictness. For example, in contrast to other policy levels, the policy level `Hash` is very fine-grained and reports any modification of an image's content. Therefore, this level is appropriate for protecting images deployed on security-critical systems. However, the policy in which this level is specified has to be updated every time the content of the file is modified. This makes `Hash` an operationally challenging policy level for verifying images that are frequently modified. Since WDAC offers a wide range of granularities at which it may verify images, the trade-off between policy manageability and verification strictness has to be tailored with respect to the behavior of the image landscape and the security-criticality of a given system protected by WDAC.

A set of relevant, interrelated considerations enable the manageable and security-efficient WDAC deployment. These considerations are:

- *definition of the scope and the goal of WDAC deployment*: WDAC enables the control over the execution of images at system-wide level. However, if image execution control at user-level, or user group-level, is needed, alternative mechanisms should be deployed in addition to WDAC. An example is Microsoft AppLocker [ms_appl];
- *identification of systems at which the deployment of WDAC is not applicable*: Not all systems can be protected by WDAC in an effective and straightforward manner. Such systems are, for example, systems used for code development. This is because they execute newly created and unknown code on a regular basis. The correct construction of a WDAC policy for deployment on such a system is a challenging and error-prone task;
- *blocking the execution of images that allow the arbitrary execution of code*: WDAC is ineffective when it comes to the blocking of execution of arbitrary code by an image that is allowed to execute by a deployed WDAC policy. This execution effectively bypasses WDAC protections. When WDAC is deployed on a given system, there should be an ongoing process for the identification of images that may execute arbitrary code;
- *management of the signing certificate and process*: A WDAC policy that is to be deployed should be digitally signed with a code signing certificate in order to prevent unauthorized policy modifications. The process of signing a WDAC policy should be conducted on a dedicated, security-hardened system so that the signing certificate and process are secure;
- *deployment of monitoring and analysis mechanisms*: The deployment of centralized monitoring and analysis mechanisms should be considered. These mechanisms should monitor and analyze logged events by the deployed WDAC policy, such as policy violations. In the case of policy violations, appropriate follow-up procedures should be enforced;
- *implementation of an image lifecycle policy*: Since WDAC controls what images are allowed to execute on a given system, the implementation of an image lifecycle policy should be considered. This policy should track any changes in the landscape of images that are deployed on the system, such as newly deployed images, or versions of images. It should also reflect these changes in the deployed WDAC policy.

Windows 10 uses the Event Tracing for Windows (ETW) framework ([ERNW WP2], Section 4.1) for logging WDAC events and events produced by the non-configurable code integrity functionalities. Table 2 provides the names and the globally unique identifiers (GUIDs) of relevant ETW providers.

ETW provider	GUID
`Microsoft-Windows-Applocker`	`CBDA4DBF-8D5D-4F69-9578-BE14AA540D22`
`Microsoft-Windows-CodeIntegrity`	`4EE76BD8-3CF4-44A0-A0AC-3937643E37A3`
`Microsoft-Windows-DeviceGuard`	`F717D024-F5B4-4F03-9AB9-331B2DC38FFB`

Table 2: ETW providers logging code integrity-related events (summarizing overview)

Evaluation summary The non-configurable and configurable (WDAC) code integrity functionalities successfully prevent the execution of untrusted code. By implementing the concept of policies in which users may set options and specify rules, WDAC is highly configurable and effective. It offers a wide range of granularities at which the trustworthiness of code may be verified. This makes WDAC suitable for protecting a variety of systems. However, the deployment of WDAC requires considerable preparation and policy management efforts. This is crucial for the operationally manageable, practically feasible, and security-efficient WDAC deployment. The security of the code integrity features of Device Guard and Windows 10 is solid. They are implemented as part of the boot manager, the Windows loader, the kernel, and external library files, whose integrity is verified. However, WDAC exposes many configuration points to system users, including points for configuring the criteria based on which the trustworthiness of images is verified. Examples are the policy options `Enabled: Managed Installer` and `Enabled: Intelligent`

`Security Graph Authorization`. This opens possibilities for compromising WDAC, such as bypassing WDAC protections. Some of these policy options delegate verification of images to other entities, such as `AppLocker` or the `Intelligent Security Graph` [ms_inteli]. Therefore, the implementation of the user-configurable WDAC features, and the impact that these features have on image verification, are not part of this work package and remain to be analyzed in detail.

1.3 Concepts and Terms

The Device Guard component of Windows 10 implements a feature for preventing the execution of untrusted code. Untrusted code is program code whose integrity and authenticity cannot be verified. For example, this is code that has been tampered with in an unauthorized manner, or originates from untrusted sources. Device Guard implements a feature referred to as configurable code integrity. Configurable code integrity takes user-defined criteria into account in order to verify images, that is, to allow only specific images – executable files – to execute [ms_dg]. These criteria may involve cryptographic information (e.g., hash values) or non-cryptographic information (e.g., file names).

In addition to configurable code integrity, Windows 10 implements code integrity functionalities that do not take user-defined criteria into account. These are implemented as part of the Windows boot manager, the Windows loader, and the kernel ([ERNW WP5], Section 2.2). This work refers to these functionalities as non-configurable code integrity.

When enabled, the virtual secure mode (VSM) feature – hypervisor code integrity (HVCI), protects configurable and non-configurable code integrity functionalities by executing them in the secure environment ([ERNW WP6], Section 1.3). If the Unified Extensible Firmware Interface (UEFI) is present, the UEFI SecureBoot feature may be deployed for the verification of the integrity of the UEFI firmware and the Windows boot entities, the boot manager and the Windows loader ([ERNW WP2], Section 3.5), ([ERNW WP5], Section 2.2).

The configurable code integrity features can be structured into two categories: user-mode code integrity (UMCI) and kernel-mode code integrity (KMCI) ([Yosif 2017], Chapter 7). UMCI is for entities that operate in user-mode, such as user applications and services. KMCI is for entities that operate in kernel-mode. This includes the kernel and its extensions, such as drivers. The UMCI and KMCI implementations of the configurable code integrity feature are also known as Windows Defender Application Control (WDAC).

Figure 3 depicts a compact overview of the architecture of the Device Guard and Windows 10 code integrity features. Configurable code integrity is based on user-defined rules. Among other things, these rules may specify file names, file versions, and hashes of images. An image is verified based on comparing rule-specified data with relevant data associated with the image. For example, a rule may specify the image's hash value. When the image is verified, Windows compares the rule-specified hash value with a hash value that it has calculated. In the case of a mismatch, the image may not be allowed to execute. Section 2.2.2 provides a detailed overview of the image verification process based on user-defined rules.

User-defined rules are stored in a policy file, referred to as WDAC policy in this work (`Policy file` in Figure 3). This file is written in the Extensible Markup Language (XML) format and then converted into binary format for deployment. The WDAC policy can be digitally signed in order to prevent modifications after it is deployed. In addition, the Trusted Platform Module (TPM) measures WDAC policies for integrity measurement purposes ([ERNW WP5], Section 2.4).

An example WDAC policy in XML format is placed in the Appendix, section 'A WDAC Policy' (some file content is trimmed for compactness). This policy was automatically generated with the `New-CIPolicy` PowerShell cmdlet ([ms_newci], see Section 3.1.1.1) assuming that the executable files deployed in the system, at which the cmdlet was executed, are trusted.

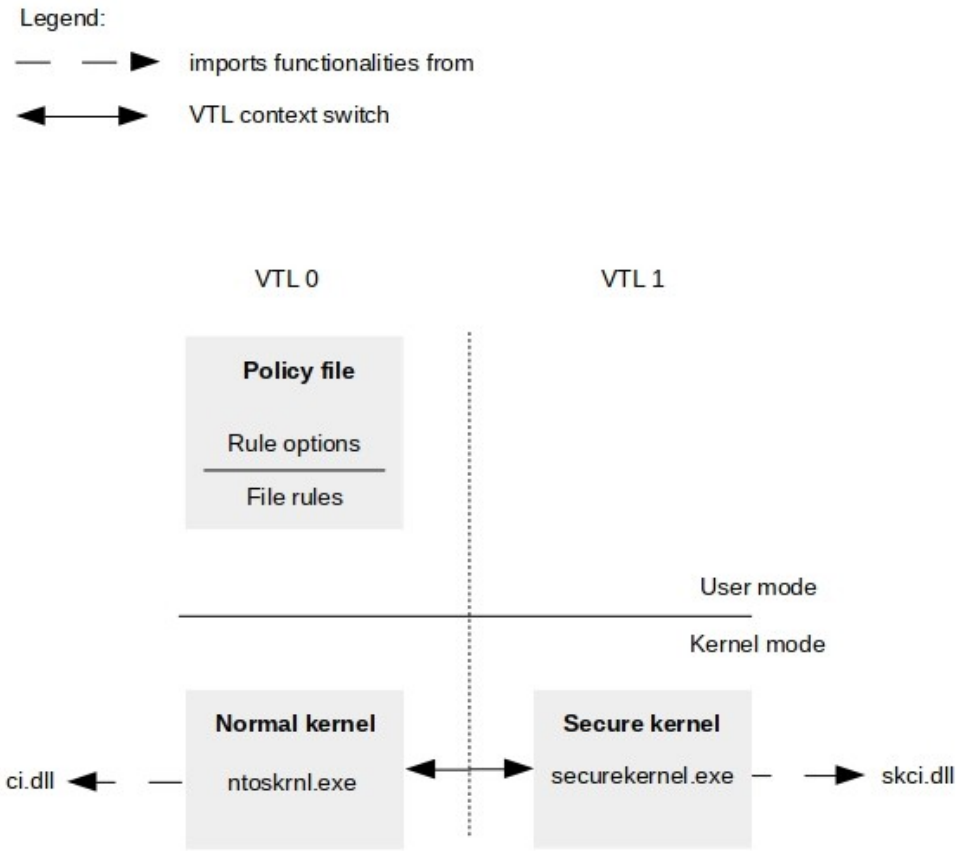

Figure 3: The architecture of the Device Guard and Windows 10 code integrity features

A WDAC policy consists of rules grouped into sections (see, for example, the `<Rules>`, `<FileRules>` and `<Signers>` tags in the Appendix, section 'A WDAC Policy'). In a WDAC policy, a user may define:

- policy rule options (`Rule options` in Figure 3): Policy rule options configure the overall functionality of WDAC. An example is the `Enabled: UMCI` option, which enables UMCI. Table 3 lists the different policy rule options and provides descriptions. The descriptions presented in Table 3 are based on the information available at [ms_rules];
- file rules (`File rules` in Figure 3): File rules configure verification for images. This configuration is done based on associating a specific level with file rules. Such levels specify at what level a given image is trusted. This work refers to these levels as policy levels. Table 4 lists the different policy levels and provides descriptions. The descriptions presented in Table 4 are based on the information available at [ms_rules].

The policy rule options and policy levels that are available on a given Windows 10 instance can be observed by investigating the policy XML schema. The schema is stored in the `Windows\schemas\CodeIntegrity\cipolicy.xsd` file. Table 3 and Table 4 present only the information about policy rule options and policy levels that is available at [ms_rules]. Section 3.1.3 provides recommendations on constructing WDAC policies. This involves specifying policy rule options and policy levels discussed in this section.

The policy levels make WDAC highly configurable and allow for administrators to decide on a trade-off between policy manageability and verification strictness. For example, in contrast to `FileName`, the policy level `Hash` (see Table 4) reports any modification of a file's content. However, the policy in which this level is specified has to be updated every time the content of the file is modified. This makes `Hash` an operationally challenging policy level for verifying files that are frequently modified.

Policy rule option	Description
Enabled: **UMCI**	This option applies the deployed WDAC policy to entities that operate in user- and in kernel-mode. By default, a WDAC policy applies only to entities that operate in kernel-mode.
Enabled: Boot Menu Protection	Currently not supported.
Required: **WHQL**	This option requires every driver specified in the WDAC policy to be signed by the Windows Hardware Quality Labs (WHQL). WHQL signs images that have passed Windows Hardware Certification Kit (WHCK) tests [ms_whlk] [ms_whqlsig].
Enabled: Audit Mode	This option enables the audit mode of a WDAC policy – the option allows for all images to execute, but logs relevant events. By default, a WDAC policy operates in audit mode. If this policy rule option is not set, a WDAC policy operates in enforcement mode – images whose execution is not allowed by the policy are not executed.
Disabled: Flight Signing	This option restricts the execution of images that are flight signed. Flight signed images are images that are signed during their development. Flight signed images are typically release candidates, for example, Windows Insider Preview image builds [ms_insider].
Enabled: Inherit Default Policy	Currently not supported.
Enabled: Unsigned System Integrity Policy	This option allows the WDAC policy to be deployed unsigned. By default, a WDAC policy has to be signed.
Allowed: Debug Policy Augmented	Currently not supported.
Required: EV Signers	This option requires for driver images to be signed by the WHQL and by Extended Validation (EV) certificates. For an EV certificate to be issued to a given entity, the entity is subjected to a rigorous vetting by a certificate authority.
Enabled: Advanced Boot Options Menu	This option configures the Windows advanced boot menu to be presented to physically present users when a WDAC policy is deployed. By default, this menu is not presented.
Enabled: Boot Audit on Failure	This option configures the WDAC policy operating in enforcement mode to switch to audit mode if image verification fails during system startup.
Disabled: Script Enforcement	Currently not supported.
Required: Enforce Store Applications	This option applies the WDAC policy to Universal Windows Applications (UWAs). Otherwise, WDAC policies are not applied to UWAs.
Enabled: Managed Installer	This option allows for images installed by a software distribution solution, such as the System Center Configuration Manager [ms_sscm], to execute. This option is not available on the Windows 10 system that is target for analysis.

Policy rule option	Description
`Enabled: Intelligent Security Graph Authorization`	This option allows for images classified as “known good” by the `Intelligent Security Graph` [ms_inteli] to execute. This option is not available on the Windows 10 system that is target for analysis.
`Enabled: Invalidate EAs on Reboot`	This option invalidates cached image classifications by the `Intelligent Security Graph` [ms_inteli] on system reboot. Therefore, it forces re-evaluation of images that have been allowed to execute with the `Enabled: Intelligent Security Graph Authorization` option configured. This option is not available on the Windows 10 system that is target for analysis.
`Enabled: Update Policy No Reboot`	This option allows for modifications to an already deployed WDAC policy to be applied without system reboot. By default, for changes to a deployed WDAC policy to take effect, the system at which the policy is deployed has to be rebooted. This option is not available on the Windows 10 system that is target for analysis.

Table 3: Policy rule options

Policy level	Description
`Hash`	This level verifies an image based on the image’s hash value.
`FileName`	This level verifies an image based on the image’s name. This name is stored as part of the image as an image property (see Section 2.2.2).
`LeafCertificate`	This level verifies an image based on a hash value of a portion of the certificate (see Section 2.2.2) issued to the image’s signer. This certificate is the leaf of the certificate chain used to sign the image.
`PcaCertificate`	This level verifies an image based on a hash value of a portion of the certificate (see Section 2.2.2) that is at the highest position in the certificate chain used to sign the image, with the exception of the root certificate. This is the certificate below the root certificate in the certificate chain. It is referred to as the PCAcertificate in this work.
`RootCertificate`	Currently not supported.
`Publisher`	This level verifies an image based on a hash value of a portion of the PCAcertificate (see Section 2.2.2) and the common name (CN) field of the leaf certificate in the certificate chain used to sign the image. This level is a combination of the `PcaCertificate` level with a verification based on the previously mentioned CN field.
`SignedVersion`	This level verifies an image based on a hash value of a portion of the PCAcertificate (see Section 2.2.2), the CN field of the leaf certificate in the certificate chain used to sign the image, and the image’s file version. The image’s file version has to be at, or above, a minimum version specified in the WDAC policy. This level is a combination of the `Publisher` level with a verification based on the image’s file version.

Policy level	Description
`FilePublisher`	This level verifies an image based on its name, a hash value of a portion of the PCAcertificate (see Section 2.2.2), the common name **(CN)** field of the leaf certificate in the certificate chain used to sign the image, and the image's file version. This level is a combination of the `SignedVersion` level with a verification based on the image's name.
`WHQL`	This level allows an image to execute if it has been signed by the WHQL.
`WHQLPublisher`	This level allows an image to execute if it has been signed by the WHQL and verified based on the CN field of the leaf certificate in the certificate chain used to sign the image. This level is a combination of the `WHQL` level with a verfication based on the previously mentioned CN field.
`WHQLFilePublisher`	This level allows an image to execute if it has been signed by the WHQL, verified based on the CN field of the leaf certificate in the certificate chain used to sign the image, and verified based on the image's file version. The image's file version has to be at, or above, a minimum version specified in the WDAC policy. This level is a combination of the `WHQLPublisher` level with a verfication based on the image's file version.

Table 4: Policy levels

Once a WDAC policy in XML format is converted into binary format, it can be deployed. For example, the group policy at the `Administrative Templates\System\Device Guard` policy path may be used for policy deployment. Windows 10 stores WDAC policies in the `SIPolicy.p7b` file. On non-UEFI platforms, Windows 10 places the `SIPolicy.p7b` file in the `%System%\System32\CodeIntegrity\` directory. On UEFI-based platforms, Windows 10 places the `SIPolicy.p7b` file additionally in the `\EFI\Microsoft\Boot\` directory of the boot partition ([ERNW WP5], Section 2.2.1).

Figure 4 depicts the placement of a WDAC policy that is stored in the binary file `C:\Users\ernw\Desktop\DeviceGuardPolicy.bin`. This file is deployed by configuring the `Administrative Templates\System\Device Guard` group policy with the `Group Policy Object Editor` utility. Once a user configures this group policy, the `Group Policy Object Editor` utility loads the `dggpext.dll` library file and invokes the `InstallConfigCIPolicy` function. This function copies the content of `DeviceGuardPolicy.bin` to the `%System%\System32\CodeIntegrity\SIPolicy.p7b` and the `\EFI\Microsoft\Boot\SIPolicy.p7b` file, depending on the presence of UEFI. The analysis presented in this work was conducted on a platform where *UEFI* is not present.

The configurable and non-configurable code integrity features implement functionalities in the boot manager, the Windows loader, and the Windows kernel. In the context of the boot manager and the Windows loader, code integrity functionalities are implemented as part of their executables. In the context of the Windows kernel, code integrity functionalities are implemented as kernel routines in external library files. If the VSM feature HVCI is disabled, code integrity functionalities are executed in the context of the `ci.dll` library file. This file is loaded by the `ntoskrnl.exe` executable, which implements the normal kernel (`Normal kernel` in Figure 3, ([ERNW WP6], Section 1.3). The `ci.dll` library file exposes an interface of functions to the kernel for use. Section 2.2.1.4 and Section 2.1.1.1 provide an overview of the kernel loading `ci.dll` and of the functionalities that `ci.dll` exposes to the kernel for use.

Figure 4: Placement of a WDAC policy

If HVCI is enabled, Windows routes code integrity functionalities to the secure environment, that is, to the virtual trust level (VTL) 1, for execution (`VTL 0`, `VTL 1`, and `VTL context switch` in Figure 3). Code integrity functionalities are then executed in the context of the `skci.dll` library file. This prevents attackers that have gained access to the normal environment to tamper with code integrity functionalities. `skci.dll` is loaded by the `securekernel.exe` executable, which implements the secure kernel (`Secure kernel` in Figure 3, ([ERNW WP6], Section 1.3). Section 2.1.1.2 provides an overview of the functionalities that are implemented in `skci.dll` and are exposed to the secure kernel for use. It also discusses the way in which code integrity functionalities are routed from the normal to the secure environment.

2 Technical Analysis of Functionalities

2.1 Non-Configurable Code Integrity

The non-configurable code integrity feature verifies code integrity based on the Authenticode digital signing technology ([ERNW WP5], Section 2.2.3). The integrity of an image is verified based on a verification of the Authenticode signature associated with the image. An Authenticode signature is a data structure that stores data relevant for integrity verification. This includes the certificate chain used to sign the image, a hash value of the image, an encrypted digest of the hash value, and the hashing algorithm used for calculating the hash value. The Microsoft Windows Authenticode Portable Executable Signature Format Specification [ms_auth] provides a detailed description of the content and the format of the Authenticode signature. This section discusses only specific data stored in this signature, which is relevant to the discussions in this work. Figure 5 depicts a compact overview of the Authenticode signature.

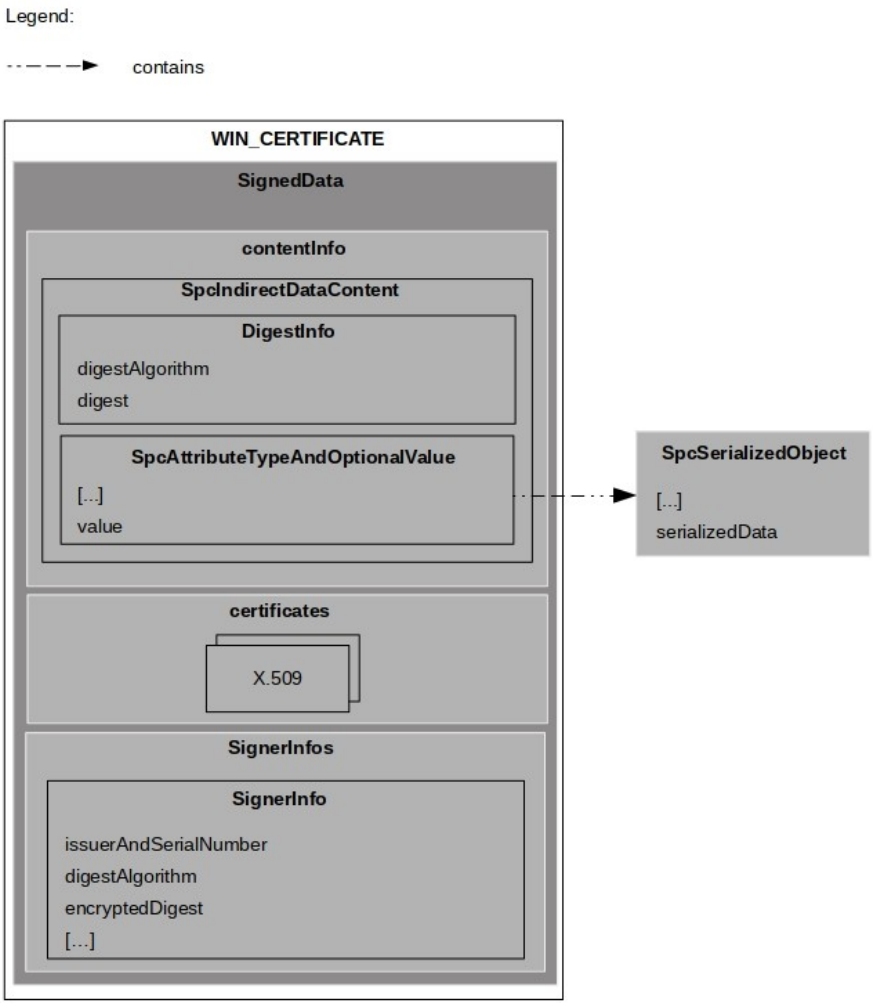

Figure 5: A compact overview of the Authenticode signature

The Authenticode signature is stored in a structure named `SignedData`, placed within a `WIN_CERTIFICATE` structure. `SignedData` stores several data fields and structures. This work focuses on the structures `contentInfo`, `certificates`, and `SignerInfos`, which are stored in `SignedData`.

`ContentInfo` stores a structure named `SpcIndirectDataContent`. This structure stores another structure named `DigestInfo`, which contains the fields `digestAlgorithm` and `digest`. `digest` stores

a hash value of the image associated with the Authenticode signature. ([ms_auth], Section 'Calculating the PE Image Hash' describes how this value is calculated). This work refers to the hash value of the image stored in `digest` as file hash. `digestAlgorithm` specifies the hash algorithm used to calculate the file hash.

In addition to `DigestInfo`, `SpcIndirectDataContent` stores a structure named `SpcAttributeTypeAndOptionalValue`. This structure ultimately references a structure named `SpcSerializedObject`. This structure stores in its field `serializedData` hash values of pages of the image associated with the Authenticode signature. This work refers to these hash values as page hashes. Windows 10 may verify the integrity of a given image based on the image's file hash and/or the image's page hashes. The page hashes are optional and may not be present in an Authenticode signature.

The `certificates` structure stores the certificate chain used to sign the image associated with the Authenticode signature. This includes the certificate of the image's signer and any intermediate certificates. The certificate of the image's signer is the leaf certificate in the certificate chain (see Table 4). The certificates are stored in the X.509 version 3 format.

The `SignerInfos` structure contains a single structure named `SignerInfo` ([ms_auth], Section 'SignerInfo'). Among other things, `SignerInfo` stores values that describe the certificate of the image's signer and an encrypted digest of the `ContentInfo` structure. Some fields of `SignerInfo` are `issuerAndSerialNumber` (the issuer and the serial number of the signing certificate), `encryptedDigest` (a signed hash of `ContentInfo`) and `digestAlgorithm` (the hash algorithm used to calculate the hash value of `ContentInfo`). In accordance with the Authenticode signing technology, Windows 10 verifies first the integrity of the Authenticode signature itself (i.e., it verifies the integrity of the `ContentInfo` structure). It then extracts the file hash or the page hashes from `ContentInfo` for image integrity verification purposes ([ERNW WP5], Section 2.2.3), ([ms_auth], Section 'SignerInfo'). The integrity of the Authenticode signature is verified by verifying the encrypted digest of `ContentInfo` stored in `SignerInfo`.

An Authenticode signature associated with a given image may be embedded in the image. ([ERNW WP5], Section 2.2.3) describes the way in which Authenticode signatures are embedded. Alternatively, an Authenticode signature may be detached from the image with which it is associated. Windows 10 uses catalogs to associate detached Authenticode signatures with a given image. Catalogs are files that contain a set of file hashes such that each hash identifies a specific image. The catalog file itself is signed with an embedded Authenticode signature. Therefore, a single catalog file serves as a detached signature that may be associated with multiple images [ms_embcat].

2.1.1 Overview of Functionalities

This section provides a structured overview of the code integrity functionalities implemented in `ci.dll` and `skci.dll`. These library files are loaded by the normal and the secure kernel and provide code integrity services to them (see Section 1.3). The focus of this section is on structuring the code integrity functionalities that `ci.dll` and `skci.dll` expose in the form of exported functions. These functions act as code integrity services exposed by `ci.dll` and `skci.dll` to the normal and the secure kernel, respectively.

This section structures code integrity functionalities implemented in exported functions based on:

- the documentation about some of these functions provided in [Mi2018]; and
- a non-detailed, brief technical analysis of the implementations of these functions. A detailed technical analysis of the implementation of each function exported by `ci.dll` and `skci.dll` is out of the scope of this work. Detailed descriptions of the functionalities implemented in some functions exported by `ci.dll` and `skci.dll` are provided as part of the technical analysis presented in this work.

The structured overviews presented in this section are general categorizations for orientation purposes. Code integrity functionalities implemented in `ci.dll` and `skci.dll` are categorized based on the following criteria:

- *initialization:* To this category belong functionalities that are related to the initialization of code integrity functionalities. This includes the execution of self-tests for evaluating functional correctness and the integrity of the implementations of the code integrity functionalities themselves;
- *image integrity verification*: To this category belong functionalities that are related to image integrity verification. Such functionalities can be further categorized based on the verified image sections (category *image sections*) and on the placement of the Authenticode signature used for image integrity verification (category *signature placement*, see Section 2.1):
  - *image sections*: This category can be split into the following sub-categories:
    - *page*: To this category belong functionalities that are related to image integrity verification based on page hashes (see Section 2.1);
    - *file*: To this category belong functionalities that are related to image integrity verification based on file hashes (see Section 2.1);
  - *signature placement*: This category can be split into the following sub-categories:
    - *embedded*: To this category belong functionalities that are related to image integrity verification based on embedded Authenticode signatures (see Section 2.1);
    - *catalogs*: To this category belong functionalities that are related to image integrity verification based on catalogs, that is, detached Authenticode signatures (see Section 2.1);
- *miscellaneous*: To this category belong functionalities that do not fit into any of the categories above. Such functionalities perform activities such as memory allocation and deallocation, status and information querying, and cryptographic operations, such as hashing.

2.1.1.1 Functionalities of ci.dll

`ci.dll` statically exports 8 functions: `CiCheckSignedFile`, `CiFindPageHashesInCatalog`, `CiFindPageHashesInSignedFile`, `CiFreePolicyInfo`, `CiGetPEInformation`, `CiInitialize`, `CiValidateFileObject`, and `CiVerifyHashInCatalog`. However, when initialized, it exports additional 18 functions in the form of callback functions. Once they are exported, the normal kernel can invoke these functions. The exported callback functions are: `CiValidateImageHeader`, `CiValidateImageData`, `CiQueryInformation`, `CiSetFileCache`, `CiGetFileCache`, `CiHashMemory`, `KappxIsPackageFile`, `CiCompareSigningLevels`, `CiValidateFileAsImageType`, `CiRegisterSigningInformation`, `CiUnregisterSigningInformation`, `CiInitializePolicy`, `SIPolicyQueryPolicyInformation`, `CiValidateDynamicCodePages`, `SIPolicyQuerySecurityPolicy`, `CiGetStrongImageReference`, `CiReleaseContext`, and `CiHvciSetImageBaseAddress`.

The kernel invokes the `SepInitializeCodeIntegrity` and `CiInitialize` functions (see Section 2.2.1.4) in order to initialize code integrity. This function exports the previously mentioned callback functions. Figure 6 depicts several of these functions. The functions depicted in Figure 6 are extracted from the execution context of the normal kernel. Figure 7 depicts the kernel invoking the `CiInitializePolicy` callback function after the function has been exported to it.

The table in the Appendix, section 'Functionalities Exported by ci.dll', categorizes the functionalities of the functions exported by `ci.dll` (column 'Function') into the categories defined in Section 2.1.1 (column 'Category'). The `*` symbol marks a category applying to a function. The categorized functionalities may be executed conditionally – the functions exported by `ci.dll` may not always perform them, since their execution depends on the conducted integrity verification scenario.

```
kd> .for (r $t0=1;@$t0<=18;r $t0=@$t0+1){ ln poi(nt!SeCiCallbacks+8*@$t0) }
[...]
(fffff800`d2ef4a50)  CI!CiSetFileCache  |  (fffff800`d2ef4eb4)   CI!CipGetFileCache
Exact matches:
    CI!CiSetFileCache (<no parameter info>)
[...]
(fffff800`d2ef51d0)  CI!CiGetFileCache  |  (fffff800`d2ef5220)   CI!CiGetCatalogHint
Exact matches:
    CI!CiGetFileCache (<no parameter info>)
[...]
(fffff800`d2ef2c70)  CI!CiQueryInformation  |  (fffff800`d2ef2e1c)   CI!CipCheckConfigOptions
Exact matches:
    CI!CiQueryInformation (<no parameter info>)
[...]
(fffff800`d2efb400)  CI!CiValidateImageHeader  |  (fffff800`d2efbe10)   CI!CiValidateImageData
Exact matches:
    CI!CiValidateImageHeader (<no parameter info>)
[...]
(fffff800`d2efbe10)  CI!CiValidateImageData  |  (fffff800`d2efbf00)   CI!CiValidateDynamicCodePages
Exact matches:
    CI!CiValidateImageData (<no parameter info>)
[...]
```

Figure 6: Code integrity callback functions

```
kd> bp CI!CiInitializePolicy
[...]
Breakpoint 2 hit
CI!CiInitializePolicy:
fffff800`d2ef2130 48895c2420      mov     qword ptr [rsp+20h],rbx
kd> kc
 # Call Site
00 CI!CiInitializePolicy
01 nt!SeCodeIntegrityInitializePolicy
02 nt!Phase1InitializationDiscard
03 nt!Phase1Initialization
04 nt!PspSystemThreadStartup
05 nt!KiStartSystemThread
```

Figure 7: The kernel invoking the exported callback function CIInitializePolicy (the prefix CI! indicates that CIInitializePolicy is implemented in ci.dll)

2.1.1.2 Functionalities of skci.dll

`skci.dll` statically exports 9 functions. These functions are: `SkciCreateCodeCatalog`, `SkciCreateSecureImage`, `SkciFinalizeSecureImageHash`, `SkciFinishImageValidation`, `SkciFreeImageContext`, `SkciInitialize`, `SkciTransferVersionResource`, `SkciValidateDynamicCodePages`, and `SkciValidateImageData`.

The table in the Appendix, section 'Functionalities Exported by skci.dll', categorizes the functionalities of the functions exported by `skci.dll` (column 'Function') into the categories defined in Section 2.1.1 (column 'Category'). The ✱ symbol marks a category applying to a function. The categorized functionalities may be executed conditionally – the functions exported by `skci.dll` may not always perform them, since their execution depends on the conducted integrity verification scenario.

The functions exported by `skci.dll` are invoked when Windows 10 routes code integrity functionalities from the normal to the secure kernel (see Section 1.3). This section provides an overview of the invocation paths of the functions exported by `skci.dll`, from the triggering of their invocation in the context of the

normal environment (paragraph 'Normal environment), to their execution in the context of the secure environment (paragraph 'Secure environment).

Secure environment In the context of the secure kernel, the functions exported by `skci.dll` are primarily invoked by functions with the prefix `Skm` (an exception is the `SkInitSystem` function). The column 'skci.dll' of the table in the Appendix, section 'ci.dll and skci.dll: Invocation Paths', lists the functions exported by `skci.dll`. The column 'Secure kernel' of this table lists the functions that invoke the functions exported by `skci.dll` – functions with prefix `Skm` and `SkInitSystem`.

The functions with prefix `Skm` and `SkInitSystem` are primarily invoked when the secure kernel processes specific secure services ([ERNW WP6], Section 2.2.3). `SkinitSystem` is also invoked during the initialization of the secure kernel, by the `SkiSystemStartupFunction`. Secure services are requested by the normal environment. They can be uniquely identified by their secure service call numbers (SSCNs). The column 'SSCN' in the table in the Appendix, section 'ci.dll and skci.dll: Invocation Paths', lists the SSCNs of the secure services that execute the functions with prefix `Skm`, or `SkInitSystem`. These functions are invoked in the `IumInvokeSecureService` function, which is where secure service requests are processed ([ERNW WP6], Section 2.2.3). `SkmiDeleteImage` is invoked through a dynamic function pointer. Therefore, this function cannot be associated with a specific SSCN based on static analysis only (`/` in the table in the Appendix, section 'ci.dll and skci.dll: Invocation Paths').

Normal environment In order to invoke functions implemented in `skci.dll`, the normal kernel and functions implemented in `ci.dll` request secure services ([ERNW WP6], Section 2.2.3). In order to request a secure service, they use the `g_CiVslHvciInterface` variable implemented in `ci.dll`. This variable stores pointers to functions implemented in the normal kernel. These functions have names with the prefix `Vsl`. They invoke the `VslpEnterIumSecureMode` function. This function requests secure services from the secure kernel by issuing VTL calls ([ERNW WP6], Section 2.2.3). The SSCNs of requested secure services are stored as the second parameter of `VslpEnterIumSecureMode`. Table 5 lists the positions, or offsets, in the `g_CiVslHvciInterface` variable (column 'g_CiVslHvciInterface position') at which pointers to functions with the prefix `Vsl` are stored (column 'Function'). The column 'SSCN' of Table 5 lists the SSCNs identifying the secure services requested by the functions with the prefix `Vsl`.

g_CiVslHvciInterface position	Function	SSCN
`g_CiVslHvciInterface`	`VslCreateSecureAllocation`	`0x13`
`g_CiVslHvciInterface + 0x8`	`VslFillSecureAllocation`	`0x14`
`g_CiVslHvciInterface + 0x10`	`VslMakeCodeCatalog`	`0x15`
`g_CiVslHvciInterface + 0x18`	`VslCreateSecureImageSection`	`0x16`
`g_CiVslHvciInterface + 0x20`	`VslValidateSecureImagePages`	`0xC1`
`g_CiVslHvciInterface + 0x28`	`VslFinalizeSecureImageHash`	`0x17`
`g_CiVslHvciInterface + 0x30`	`VslFinishSecureImageValidation`	`0x18`
`g_CiVslHvciInterface + 0x38`	`VslPrepareSecureImageRelocations`	`0x19`
`g_CiVslHvciInterface + 0x40`	`VslRelocateImage`	`0x1A`
`g_CiVslHvciInterface + 0x48`	`VslCloseSecureHandle`	`0x1B`
`g_CiVslHvciInterface + 0x50`	`VslGetNestedPageProtectionFlags`	`0xE7`
`g_CiVslHvciInterface + 0x58`	`VslValidateDynamicCodePages`	`0x1C`
`g_CiVslHvciInterface + 0x60`	`VslTransferSecureImageVersionResource`	`0x1D`

Table 5: Functions referenced by g_CiVslHvciInterface

The invocation of the functions referenced by `g_CiVslHvciInterface` is protected by ControlFlowGuard ([ERNW WP2], Section 3.7.4). Figure 8 depicts an invocation of the function referenced at offset `0x20` of `g_CiVslHvciInterface` – `VslValidateSecureImagePages` (see Table 5). In

accordance with the design of ControlFlowGuard, the `rax` register, at the time the `_guard_dispatch_icall_fptr` function is invoked, points to the function that is ultimately invoked. Therefore, the places where the functions referenced by `g_CiVslHvciInterface` are invoked can be identifed by searching for invocations of `_guard_dispatch_icall_fptr` in the implementations of `ci.dll` and the normal kernel, such that the `rax` register points at a given offset of `g_CiVslHvciInterface`.

```
CI!CiHvciValidateImageData+0x93:
fffff80d`c6c16643 b900000000      mov     ecx,0
fffff80d`c6c16648 488bf3          mov     rsi,rbx
[...]
fffff80d`c6c1668b 488b05ce35feff  mov     rax,qword ptr [CI!g_CiVslHvciInterface+0x20 (fffff80d`c6bf9c60)]
fffff80d`c6c16692 03d6            add     edx,esi
[...]
fffff80d`c6c166a1 ff15c98ffeff    call    qword ptr [CI!_guard_dispatch_icall_fptr (fffff80d`c6bff670)]
fffff80d`c6c166a7 428b94f59c000000 mov     edx,dword ptr [rbp+r14*8+9Ch]
[...]
```

Figure 8: ControlFlowGuard protecting functions referenced by `g_CiVslHvciInterface`

A brief analysis revealed that most of the functions referenced by `g_CiVslHvciInterface` are invoked by functions with the prefix `CiHvci`. These functions are implemented in `ci.dll`. The functions with the prefix `CiHvci` are primarily invoked through dynamic function pointers. This makes the identification of the functions invoking them challenging. Figure 9 depicts an example of invocation of a function with prefix `CiHvci`. It depicts the `CiValidateImageData` function conditionally invoking `CiHvciValidateImageData`. The identification of the functions invoking functions with the prefix `CiHvci` is out of the scope of this work.

```
kd> uf CiValidateImageData
CI!CiValidateImageData:
[...]
CI!CiValidateImageData+0x43:
fffff80d`c6c0be53 a802            test    al,2
fffff80d`c6c0be55 747d            je      CI!CiValidateImageData+0xc4 (fffff80d`c6c0bed4)  Branch

CI!CiValidateImageData+0x47:
fffff80d`c6c0be57 bd01000000      mov     ebp,1
fffff80d`c6c0be5c 400fb6c6        movzx   eax,sil
fffff80d`c6c0be60 40846c2460      test    byte ptr [rsp+60h],bpl
fffff80d`c6c0be65 488bcb          mov     rcx,rbx
fffff80d`c6c0be68 0f45c5          cmovne  eax,ebp
fffff80d`c6c0be6b 88442420        mov     byte ptr [rsp+20h],al
fffff80d`c6c0be6f e83ca70000      call    CI!CiHvciValidateImageData (fffff80d`c6c165b0)
[...]
```

Figure 9: Conditional invocation of CiHvciValidateImageData

The column 'ci.dll' of the table in the Appendix, section 'ci.dll and skci.dll: Invocation Paths', lists the functions with prefix `CiHvci` that ultimately trigger the execution of functions exported by `skci.dll` (the → symbol marks function invocation). The column 'Normal kernel' of this table lists functions implemented in the normal kernel that request secure services in order to trigger the execution of functions exported by `skci.dll`. Some of these functions are referenced by the `g_CiVslHvciInterface` variable and are invoked by the functions with prefix `CiHvci` (see Table 5). Others are invoked directly by the normal kernel, such as `VslCreateSecureImageSection`. The column 'SSCN' of the table in the Appendix, section 'ci.dll and skci.dll: Invocation Paths', lists the SSCNs identifying the secure services requested by the functions listed in the column 'Normal kernel' of this table. The execution of these services in the context of the secure kernel results in the execution of functions exported by `skci.dll`.

2.2 WDAC: Configurable Code Integrity

This section describes the process for initializing WDAC (Section 2.2.1). In addition, it discusses the way in which Windows 10 verifies images based on user-configured WDAC policies (Section 2.2.2). This work refers to this verification process as WDAC verification. WDAC verification is executed as part of the configurable code integrity feature (WDAC) of Device Guard (see Section 1.3).

2.2.1 WDAC Initialization

This section describes the process for initializing WDAC performed by the Windows loader and the kernel ([ERNW WP5], Section 2.2) when Windows 10 is booted (see Figure 10).

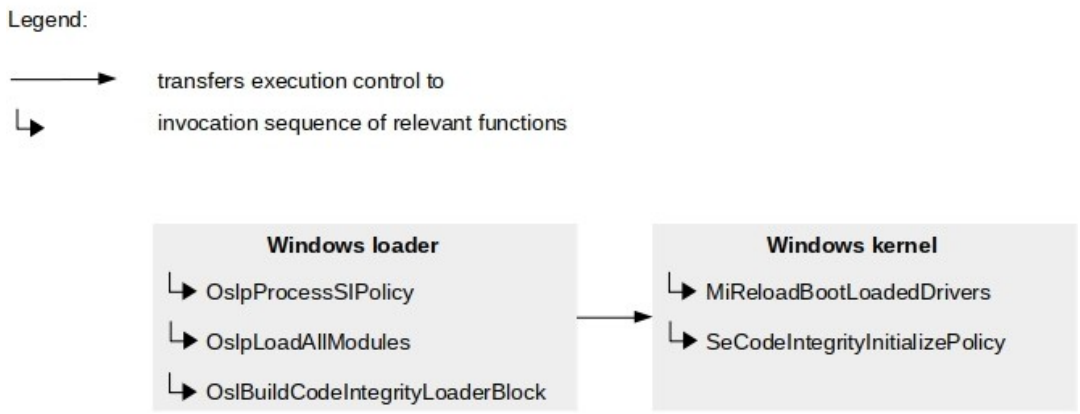

Figure 10: WDAC initialization

Windows loader The `OslPrepareTarget` function implemented as part of the Windows loader performs WDAC initialization activities. These activites are performed by the functions:

- `OslpProcessSIPolicy`;
- `OslpLoadAllModules`; and
- `OslBuildCodeIntegrityLoaderBlock`.

The functions above are invoked by `OslPrepareTarget`.

The `OslpProcessSIPolicy` function initializes and loads the WDAC policy in the context of the Windows loader. This involves verifying the integrity of the WDAC policy, if signed. Section 2.2.1.1 discusses this verification in detail. Once `OslpProcessSIPolicy` is finished executing, the WDAC policy may be used for image verification by the Windows loader. Among other images, the Windows loader verifies the integrity of the `ci.dll` file.

The functions `OslBuildCodeIntegrityLoaderBlock` and `OslpLoadAllModules` populate with *WDAC* initialization parameters the `CodeIntegrityLoaderBlock` (see Figure 17) and `LoadOrderListHead` fields ultimately referenced by the `_LOADER_PARAMETER_BLOCK` structure ([Russinovich 2012], Chapter 13). The `_LOADER_PARAMETER_BLOCK` structure is ultimately passed to the Windows kernel ([ERNW WP5], Section 2.2) at execution transfer between the Windows loader and the kernel. Section 2.2.1.2 and Section 2.2.1.3 discuss the population of `CodeIntegrityLoaderBlock` and `LoadOrderListHead` in more detail. Once `_LOADER_PARAMETER_BLOCK` is populated with WDAC initialization parameters, the Windows loader transfers the execution control to the Windows kernel. To this end, it executes the `OslArchTransferToKernel` function.

Windows kernel Once the Windows loader has transferred the execution control to the kernel, it uses the populated `_LOADER_PARAMETER_BLOCK` structure to initialize WDAC in the context of the kernel. The kernel is initialized in two phases: Phase 0 and Phase 1 ([Russinovich 2012], Chapter 13). The kernel invokes in Phase 0 the `MiReloadBootLoadedDrivers` function. This function allocates a memory region in the virtual address space assigned to the kernel for the `ci.dll` file (see Section 1.3). The starting address of this space is referred to as the image base address of `ci.dll`. Section 2.2.1.4 describes the operation of `MiReloadBootLoadedDrivers` in detail.

Once Phase 0 is finished, the kernel starts Phase 1. In this phase, the kernel continues initializing WDAC. This involves for example, invoking the `SeCodeIntegrityInitializePolicy` function, which initializes the WDAC policy. Section 2.2.1.5 describes the operation of `SeCodeIntegrityInitializePolicy` in more detail. Once `SeCodeIntegrityInitializePolicy` is finished executing, the WDAC policy may be used for image verification by the Windows kernel.

2.2.1.1 Windows Loader: OslpProcessSIPolicy

`OslpProcessSIPolicy` loads and processes the `SIPolicy.p7b` file, that is, the *WDAC* policy (see Section 1.3). If the WDAC policy is signed, `OslpProcessSIPolicy` verifies the integrity of the policy. This section discusses this verification process. The `SIPolicy.p7b` is in the (Public Key Cryptography Standards) PKCS*#7* file format [rfc_pkcs]. This format allows for specifying file-specific cryptographic data, such as digital signatures. Figure 11 depicts the Abstract Syntax Notation One (ASN.1) format of a digitally signed PKCS*#7* file. The `SignedData` data structure contains the overall data content, including related cryptographic data. This section focusses on the `digestAlgorithms`, `contentInfo`, `certificates`, and `signerInfos` fields of `SignedData`. It is important to emphasize that the PKCS*#7* file format discussed in this section is different than the format of an Authenticode signature discussed in Section 1.3. The format of Authenticode signatures however is based on the PKCS*#7* file format [ms_auth].

```
SignedData ::= SEQUENCE {
    version: Version,
    digestAlgorithms: DigestAlgorithmIdentifiers,
    contentInfo: ContentInfo,
    certificates:
       [0] ExtendedCertificatesAndCertificates,
    crls:
       [1] CertificateRevocationLists,
    signerInfos: SignerInfos
}
```

Figure 11: ASN.1 format of a PKCS#7 file

`contentInfo` stores the user-generated file rules and policy rule options in binary format (see Section 1.3). This work refers to these file rules and policy rule options as WDAC content. `OslpProcessSIPolicy` verifies the integrity of the WDAC content.

`certificates` stores the certificate chain used to sign WDAC content. The certificates are stored in the X.509 format.

`signerInfos` stores values that describe the certificate of the signer of the WDAC content, the hash value of the WDAC content, and the signed hash of the WDAC content. Some fields referenced by `signerInfos` are:

- `issuerAndSerialNumber`, which stores the issuer and the serial number of the signing certificate;
- `encryptedDigest`, which stores the signed hash of the WDAC content;
- `digestAlgorithm`, which stores the hash algorithm used to calculate the hash value of the WDAC content; and

- `authenticatedAttributes`, which stores, among other things, the hash value of the *WDAC* content.

Figure 12 depicts a portion of a signed *WDAC* policy as viewed with the `openssl` utility.

```
PKCS7:
  [...]
    contents:
      type: undefined (1.3.6.1.4.1.311.79.1)
      d.other: OCTET STRING:
        0000 - 02 00 00 00 0e 37 44 a2-c9 44 06 4c b5 51 f6   .....7D..D.L.Q.
        000f - 01 6e 56 30 76 e4 f7 07-2e 4c 19 20 4d b7 c9   .nV0v....L. M..
        001e - 6f 44 a6 c5 a2 34 00 00-c1 82 00 00 00 00 00   oD...4.........
        [...]
        012c - 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00   ...............
        013b - 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00   ...............
        014a - 00 00 00 00 00 00 03 00-00 00                  ..........

    cert:
        cert_info:
          version: 2
          serialNumber: 0x1700000002AE2B61EAE06796EA000000000002
          signature:
            algorithm: sha256WithRSAEncryption (1.2.840.113549.1.1.11)
            parameter: NULL
          issuer: DC=internal, DC=test, CN=test-WIN-97VOE5A4O8L-CA
          validity:
            notBefore: Aug  3 14:06:27 2018 GMT
            notAfter: Aug  3 14:06:27 2019 GMT
          subject: CN=testDGSigningCert
          [...]                                        .
```

Figure 12: Portion of a signed WDAC policy

`OslpProcessSIPolicy` first invokes the `BlSIPolicyCheckPolicyOnDevice` function, which invokes `BlSIPolicyReadPolicies`. `BlSIPolicyReadPolicies` loads `SIPolicy.p7b` and returns the size and ASN.1 formatted WDAC policy. The former is stored at offset `0x30`, and the latter at `0x28` of the `rsp` register (see Figure 13).

```
kd> dd poi(@rsp + 0x30) L1
00000000`001a6dc8  00000903

kd> db poi(poi(@rsp + 0x28)) L903
fffff802`26b7fee0  30 82 08 ff 06 09 2a 86-48 86 f7 0d 01 07 02 a0  0.....*.H.......
fffff802`26b7fef0  82 08 f0 30 82 08 ec 02-01 01 31 0f 30 0d 06 09  ...0......1.0...
fffff802`26b7ff00  60 86 48 01 65 03 04 02-01 05 00 30 82 01 67 06  `.H.e......0..g.
fffff802`26b7ff10  09 2b 06 01 04 01 82 37-4f 01 a0 82 01 58 04 82  .+.....7O....X..
[...]
fffff802`26b80070  00 00 03 00 00 00 a0 82-05 84 30 82 05 80 30 82  ..........0...0.
fffff802`26b80080  04 68 a0 03 02 01 02 02-13 17 00 00 00 02 ae 2b  .h.............+
fffff802`26b80090  61 ea e0 67 96 ea 00 00-00 00 00 02 30 0d 06 09  a..g........0...
fffff802`26b800a0  2a 86 48 86 f7 0d 01 01-0b 05 00 30 52 31 18 30  *.H........0R1.0
fffff802`26b800b0  16 06 0a 09 92 26 89 93-f2 2c 64 01 19 16 08 69  .....&...,d....i
fffff802`26b800c0  6e 74 65 72 6e 61 6c 31-14 30 12 06 0a 09 92 26  nternal1.0.....&
fffff802`26b800d0  89 93 f2 2c 64 01 19 16-04 74 65 73 74 31 20 30  ...,d....test1 0
fffff802`26b800e0  1e 06 03 55 04 03 13 17-74 65 73 74 2d 57 49 4e  ...U....test-WIN
fffff802`26b800f0  2d 39 37 56 4f 45 35 41-34 4f 38 4c 2d 43 41 30  -97VOE5A4O8L-CA0
fffff802`26b80100  1e 17 0d 31 38 30 38 30-33 31 34 30 36 32 37 5a  ...180803140627Z
fffff802`26b80110  17 0d 31 39 30 38 30 33-31 34 30 36 32 37 5a 30  ..190803140627Z0
fffff802`26b80120  1c 31 1a 30 18 06 03 55-04 03 13 11 74 65 73 74  .1.0...U....test
fffff802`26b80130  44 47 53 69 67 6e 69 6e-67 43 65 72 74 30 82 01  DGSigningCert0..
[...]
```

Figure 13: Loaded SIPolicy.p7b

WDAC is considered disabled if no *WDAC* policy is returned by `BlSIPolicyReadPolicies`. If a WDAC policy is returned, WDAC is considered enabled. Only users with administrative privileges can delete a WDAC policy (the `SIPolicy.p7b` file, see Section 1.3) and therefore, disable WDAC. When `BlSIPolicyReadPolicies` is finished executing, `BlSIPolicyCheckPolicyOnDevice` invokes `BlSIPolicyParsePolicyData`. This function processes the loaded WDAC policy.

Before `BlSIPolicyParsePolicyData` processes the WDAC policy, it verifies its integrity. The `MinCryptVerifySignedDataLMode` function initiates the verification of the WDAC policy. `MinCryptVerifySignedDataLMode` receives as parameters the size of the WDAC policy and the ASN.*1* formatted WDAC policy. Figure 14 depicts the invocation of `MinCryptVerifySignedDataLMode`. The integrity verification of the WDAC policy can be structured into two phases. In the first phase, the certificate of the signer of the WDAC policy is verified. In the second phase, the integrity of the WDAC policy itself is verified.

```
winload!MinCryptVerifySignedDataLMode:
00000000`007bd1e8 4055            push    rbp

kd> r
rax=00000000001a6a98 rbx=0000000000000903 rcx=fffff80226b7fee0
rdx=0000000000000903 rsi=00000000001a6bc0 rdi=000000000089c600
[...]

kd> kc
 # Call Site
00 winload!MinCryptVerifySignedDataLMode
01 winload!MinCrypL_CheckSignedData
02 winload!BlSIPolicyParsePolicyData
03 winload!BlSIPolicyCheckPolicyOnDevice
04 winload!OslpProcessSIPolicy
05 winload!OslPrepareTarget
06 winload!OslpMain
07 winload!OslMain
08 0x0
```

Figure 14: Invocation of MinCryptVerifySignedDataLMode

`MinCryptVerifySignedDataLMode` first invokes the `MinCryptVerifyCertificateWithRootInfo` function. `MinCryptVerifyCertificateWithRootInfo` verifies the certificate of the signer of the WDAC policy signer certificate against its root certificate. The verified certificate is stored in the `certificates` field of the `SignedData` structure. `MinCryptVerifyCertificateWithRootInfo` uses the root certificates embedded in the Windows loader, in the `RootTable` structure ([ERNW WP5], Section 2.2). The fact that certificates embedded in the Windows loader are used for verifying the certificate used to sign the WDAC policy shows that the root of trust for verifying the integrity of the WDAC policy is the Windows loader itself.

It is important to emphasize that in the scenario, where the WDAC policy is signed with a certificate that cannot be verified against a root certificate stored in `RootTable`, the certificate is considered valid without verification against an alternative root certificate.

Once `MinCryptVerifyCertificateWithRootInfo` is finished executing, the WDAC policy, that is, the WDAC content, is verified. To this end, `MinCryptVerifySignedDataLMode` first invokes the `MinCryptHashMemory` function. `MinCryptHashMemory` computes the hash value of the WDAC content, which stored in the `contentInfo` field of the `SignedData` structure. The algorithm used to calculate the hash value of the WDAC content is stored in `digestAlgorithms`.

`MinCryptVerifySignedDataLMode` then invokes the `I_MinCryptVerifySignerAuthenticatedAttributes` function. This function verifies the computed hash value against the hash value stored in `authenticatedAttributes`. Finally, `MinCryptVerifySignedDataLMode` invokes `MinCryptVerifySignedHash` in order to verify the signed hash of the WDAC content stored in `encryptedDigest`. To this end, it uses the previously verified signer certificate and the verified computed hash value. Only if the verifications performed by `I_MinCryptVerifySignerAuthenticatedAttributes` and `MinCryptVerifySignedHash` are successful, the WDAC content is considered authentic.

2.2.1.2 Windows Loader: OslpLoadAllModules

`OslpLoadAllModules` performs image loading and integrity verification activities. `OslpLoadAllModules` invokes `OslLoadDrivers` for loading driver executables, and `OslLoadImage` for loading any other type of image. The Windows loader loads the `ci.dll` library file in the `LoadImports` function, invoked by `OslLoadImage` (see [ERNW WP5], Section 2.2.2). All of the previously mentioned functions ultimately invoke `BlImgLoadPEImageEx`, which performs image loading and integrity verification. The image integrity verification implemented in `BlImgLoadPEImageEx` is described in [ERNW WP5] and [ERNW WP6]. Figure 15 depicts the `BlImgLoadPEImageEx` function loading `ci.dll` and its image base address (`fffff803`99b1e000`, see Section 2.2.1).

```
winload!BlImgLoadPEImageEx:
00000000`007eb9e4 488bc4          mov     rax,rsp

kd> r
[...]
r8=fffff80397feebf0  r9=00000000001a6810 r10=0000000000000000
[...]

kd> du @r8
fffff803`97feebf0  "\Windows\system32\CI.dll"

kd> dps 00000000001a6810 L1
00000000`001a6810  fffff803`99b1e000

kd> !dh -e fffff803`99b1e000
 _IMAGE_EXPORT_DIRECTORY fffff80399ba3000 (size: 00000130)
Name: CI.dll
Characteristics: 00000000 Ordinal base: 1.
Number of Functions: 11. Number of names: 8. EAT: fffff80399ba3028.
   ordinal hint target           name
         4    0 FFFFF80399B41650 CiCheckSignedFile
         5    1 FFFFF80399B41700 CiFindPageHashesInCatalog
         6    2 FFFFF80399B41780 CiFindPageHashesInSignedFile
         7    3 FFFFF80399B41790 CiFreePolicyInfo
         8    4 FFFFF80399B41520 CiGetPEInformation
         9    5 FFFFF80399B40110 CiInitialize
        10    6 FFFFF80399B4C3D0 CiValidateFileObject
        11    7 FFFFF80399B415D0 CiVerifyHashInCatalog
         1      FFFFF80399B4C9E0 [NONAME]
         2      FFFFF80399B51AA0 [NONAME]
         3      FFFFF80399B51C80 [NONAME]
```

Figure 15: The image base address of ci.dll

Once ci.dll is loaded, its image base address is stored in a linked list referenced by the LoadOrderListHead variable. This variable is stored in the _LOADER_PARAMETER_BLOCK structure (see Section 2.2.1). Figure 16 depicts a portion of _LOADER_PARAMETER_BLOCK and the LoadOrderListHead variable referencing the image base address of ci.dll.

Once the Windows loader has transferred execution control to the kernel, it uses the populated `LoadOrderListHead` variable to pass the image base address of `ci.dll` (`fffff803`99b1e000`) to the Windows kernel for allocation of `ci.dll` in kernel's context (see Section 2.2.1.4).

```
kd> dps poi(winload!OslLoaderBlock)
[...]
fffff801`ec894fd0  fffff801`ec94a6b0
[...]

kd> dl fffff801`ec94a6b0
[...]
fffff801`ec897a70  00000000`00000000 00000000`00000000
fffff801`ec898a60  fffff801`ec899a40 fffff801`ec897a60
fffff801`ec898a70  00000000`00000000 00000000`00000000
[...]

kd> dps fffff801`ec899a40 + 0x30 L1
fffff801`ec899a70  fffff803`99b1e000
```

Figure 16: A portion of _LOADER_PARAMETER_BLOCK and LoadOrderListHead

2.2.1.3 Windows Loader: OslBuildCodeIntegrityLoaderBlock

OslBuildCodeIntegrityLoaderBlock first populates the _LOADER_PARAMETER_CI_EXTENSION structure with WDAC initialization parameters. These parameters are used by the kernel to further initialize WDAC (see Section 2.2.1). A reference to _LOADER_PARAMETER_CI_EXTENSION and its size are stored in the _LOADER_PARAMETER_EXTENSION structure, in the CodeIntegrityLoaderBlockSize and the CodeIntegrityLoaderBlock, respectively (see Figure 17). The _LOADER_PARAMETER_EXTENSION structure is referenced by the Extension variable. This variable is stored in _LOADER_PARAMETER_BLOCK, at offset 0xF0 (see Figure 17).

```
typedef struct _LOADER_PARAMETER_BLOCK {
    [...]
    PLOADER_PARAMETER_EXTENSION         Extension;                          // 0xF0
    [...]
} LOADER_PARAMETER_BLOCK, * PLOADER_PARAMETER_BLOCK


typedef struct _LOADER_PARAMETER_EXTENSION {
    [...]
    PLOADER_PARAMETER_CI_EXTENSION      CodeIntegrityLoaderBlock;           // 0x9D8
    ULONG32                             CodeIntegrityLoaderBlockSize;       // 0x9E0
    [...]
} LOADER_PARAMETER_EXTENSION, * PLOADER_PARAMETER_EXTENSION;


typedef struct _LOADER_PARAMETER_CI_EXTENSION {
    [...]
    UINT8                       CodeIntegrityPolicyHash[32];                // 0x0020
    ULONG32                     CodeIntegrityPolicyType                     // 0x1338
    ULONG32                     CodeIntegrityPolicySize                     // 0x133c
    UINT8                       CodeIntegrityPolicy[CodeIntegrityPolicySize] // 0x1340
    [...]
} LOADER_PARAMETER_CI_EXTENSION, * PLOADER_PARAMETER_CI_EXTENSION;
```

Figure 17: Relevant _LOADER_PARAMETER_* structures

The `OslBuildCodeIntegrityLoaderBlock` function populates `_LOADER_PARAMETER_CI_EXTENSION` with *WDAC* initialization parameters, such as:

- `CodeIntegrityPolicyHash`: This parameter stores the hash value of the WDAC content. This hash is calculated in the the `OslpCalculateCodeIntegrityPolicyHash` function, invoked by `OslBuildCodeIntegrityLoaderBlock`;
- `CodeIntegrityPolicySize`: This parameter stores the size of the WDAC content; and
- `CodeIntegrityPolicy`: This parameter stores the WDAC content extracted from `contentInfo` (see Section 2.2.1.1).

After `OslBuildCodeIntegrityLoaderBlock` has finished executing, the Windows loader transfers the execution control to the kernel. The kernel uses the populated `_LOADER_PARAMETER_CI_EXTENSION` structure, ultimately referenced by `_LOADER_PARAMETER_BLOCK` to further initialize WDAC (see Section 2.2.1.5).

2.2.1.4 Windows Kernel: MiReloadBootLoadedDrivers

After execution control has been transferred to the kernel, it invokes the `InitBootProcessor` function. This function is responsible for conducting relevant tasks, for example, initializing memory management functionalities (see Figure 4, "*Phase 0 executive routines*", in [ERNW WP4], Section 1.3.2). `InitBootProcessor` ultimately invokes the memory management routine `MmInitSystem`. This routine, in turn, invokes `MiReloadBootLoadedDrivers`. This function allocates `ci.dll` in the context of the kernel based on the image base address of `ci.dll` (see, for example, `fffff803`99b1e000` in Figure 15), passed by the Windows loader (see Section 2.2.1.2).

`MiReloadBootLoadedDrivers` invokes the `MiUpdateThunks` function, which allocates `ci.dll` in the context of the kernel. Figure 18 depicts the invocation of `MiUpdateThunks`. The second parameter of `MiUpdateThunks` (`rdx` in Figure 18) is the image base address of `ci.dll` passed by the Windows loader (see Section 2.2.1.2), whereas the third (`r8` and `fffff808`c5fd0000` in Figure 18) is an address in the context of the kernel, where `ci.dll` is to be allocated.

Once `ci.dll` is allocated in the kernel's context, the kernel invokes the `SepInitializeCodeIntegrity` function. This function initializes the interface exposed by `ci.dll`, after which the kernel can use code integrity functionalities (see Section 2.1.1.1).

It is important to emphasize that the integrity of `ci.dll` is verified by the Windows loader (see Section 2.2.1.2). This shows that the root of trust for verifying the integrity of `ci.dll` is the Windows loader.

```
nt!MiUpdateThunks:
fffff803`9961e8c0 48895c2408      mov     qword ptr [rsp+8],rbx

kd> r
[...]
rdx=fffff80399b1e000 rsi=fffff80399b1e000 rdi=0000000000000006
 r8=fffff808c5fd0000 r9=00000000000a0000 r10=0000000000000000

kd> lm v m CI
start             end                 module name
fffff808`c5fd0000 fffff808`c6070000   CI         (deferred)
    Image path: CI.dll
    Image name: CI.dll
    Timestamp:        Tue Mar  6 06:25:49 2018 (5A9E265D)
    CheckSum:         0009D5DB
    ImageSize:        000A0000
    File version:     10.0.14393.2155
    Product version:  10.0.14393.2155
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        3.7 Driver
    File date:        00000000.00000000
    Translations:     0409.04b0
    Information from resource tables:
        CompanyName:      Microsoft Corporation
        ProductName:      Microsoft® Windows® Operating System
        InternalName:     ci.dll
        OriginalFilename: ci.dll
        ProductVersion:   10.0.14393.2155
        FileVersion:      10.0.14393.2155 (rs1_release_1.180305-1842)
        FileDescription:  Code Integrity Module
        LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

Figure 18: Relocated ci.dll file

2.2.1.5 Windows Kernel: SeCodeIntegrityInitializePolicy

After `ci.dll` has been allocated in the kernel's context and the interface exposed by it is available to the kernel, the kernel initializes the WDAC policy. The `SeCodeIntegrityInitializePolicy` function initializes the WDAC policy. This involves storage of the WDAC policy in the context of the kernel.

`SeCodeIntegrityInitializePolicy` receives as parameter the `_LOADER_PARAMETER_BLOCK` structure, populated and passed by the Windows loader (*KeLoaderBlock* in Figure 19). This structure ultimately references `_LOADER_PARAMETER_CI_EXTENSION` (*CodeIntegrityLoaderBlock* in Figure 19), which, among other things, stores `CodeIntegrityPolicy` and `CodeIntegrityPolicyHash`. `CodeIntegrityPolicy` stores the WDAC content itself (see Section 2.2.1.1).

`SeCodeIntegrityInitializePolicy` invokes the `CiInitializePolicy` function. This function receives the `_LOADER_PARAMETER_CI_EXTENSION` structure as parameter. `CiInitializePolicy` populates the `ci.dll` variables `g_SiPolicyHandles` and `g_SiPolicyHash` with the values stored in the `CodeIntegrityPolicy` and `CodeIntegrityPolicyHash` variables, respectively. An analysis of the WDAC initialization functionalities showed that the hash value stored in `CodeIntegrityPolicyHash` is not used for verifying the integrity of the WDAC content stored in `CodeIntegrityPolicy`.

Figure 19 depicts a portion of a populated `g_SiPolicyHandles` variable. Once `g_SiPolicyHandles` is populated, the Windows kernel can use the *WDAC* content stored in `g_SiPolicyHandles` for verification purposes (see Section 2.2.2). The description of each of the fields of `g_SiPolicyHandles` is out of the scope of this work.

It is important to emphasize that the integrity of the *WDAC* content is verified by the Windows loader (see Section 2.2.1.1). This shows that the root of trust for verifying the integrity of the *WDAC* content is the Windows loader.

```
SeCodeIntegrityInitializePolicy(KeLoaderBlock)
{
    [...]
    Extension = *(_LOADER_PARAMETER_EXTENSION *)(KeLoaderBlock + 0xF0);
    [...]
    CodeIntegrityLoaderBlock = *(_LOADER_PARAMETER_CI_EXTENSION *)(Extension + 0x9D8)
    [...]
    if ( CiInitializePolicy )
    {
        [...]
        CiInitializePolicy(CodeIntegrityLoaderBlock, [...]);
        [...]
    }
    [...]
}
```

Figure 19: Pseudo-code of the implementation of SeCodeIntegrityInitializePolicy

```
ci!g_SiPolicyHandles
   +0x000 PlatformID                : nt!_GUID
   +0x010 PolicyTypeID              : nt!_GUID
   [...]
   +0x02c RuleOptionFlags           : Uint4B
   [...]
   +0x068 CodeIntegrityPolicySize   : Int4B
   +0x06c CodeIntegrityPolicy       : Ptr64 Void
```

Figure 20: g_SiPolicyHandles

2.2.2 WDAC Verification

Windows 10 performs WDAC verification in the `CiEvaluatePolicyInfo` and `CipApplySIPolicyUMCI` functions, implemented in `ci.dll`. Both functions ultimately invoke the `ci.dll` function `SIPolicyValidateImage`. Figure 22 and Figure 21 depict sample function call stacks resulting in the invocation of `SIPolicyValidateImage` by `CiEvaluatePolicyInfo` and `CipApplySIPolicyUMCI`.

```
Breakpoint 0 hit
CI!SIPolicyValidateImage:
fffff80a`76d9f084 4c894c2420      mov     qword ptr [rsp+20h],r9
kd> kc
 # Call Site
00 CI!SIPolicyValidateImage
01 CI!CipApplySiPolicyEx
02 CI!CiEvaluatePolicyInfo
[...]
06 CI!CipValidateImageHash
07 CI!CiValidateImageHeader
[...]
19 nt!Phase1Initialization
1a nt!PspSystemThreadStartup
1b nt!KiStartSystemThread
```

Figure 22: Function stack: Invoking SIPolicyValidateImage by CiEvaluatePolicyInfo

```
Breakpoint 1 hit
CI!SIPolicyValidateImage:
fffff803`c3c3f084 4c894c2420      mov     qword ptr [rsp+20h],r9
kd> kc
 # Call Site
00 CI!SIPolicyValidateImage
01 CI!CipApplySiPolicyEx
02 CI!CipApplySIPolicyUMCI
03 CI!CipValidateImageHash
[....]
08 nt!MmCreateSection
[....]
11 ntdll!LdrLoadDll
[....]
```

Figure 21: Function stack: Invoking SIPolicyValidateImage by CipApplySiPolicyUMCI

All digitally signed critical system images, which include the Windows kernel and drivers loaded during the Windows boot process, are subjected to non-configurable code integrity verification when loaded. The non-configurable integrity verification of these images is documented in ([ERNW WP5], Section 2.2.3). If the non-configurable code integrity verification succeeds, WDAC verification is performed. This is done by invoking `CiEvaluatePolicyInfo` which invokes `SIPolicyValidateImage`. WDAC verification takes place only if WDAC is enabled (i.e., if a WDAC policy is deployed, see Section 2.2.1). If the non-configurable or the WDAC verification fails, the verified image is not loaded. All critical system images are digitally signed by Microsoft.

All other digitally signed images, such as third-party images operating in user-mode, are also subjected to non-configurable code integrity verification when loaded ([ERNW WP5], Section 2.2.3). After non-configurable integrity verification, WDAC verification is performed. This is done by conditionally invoking `CiEvaluatePolicyInfo` and/or `CipApplySIPolicyUMCI`. which invoke `SIPolicyValidateImage`. WDAC verification takes place only if WDAC is enabled (i.e., if a WDAC policy is deployed, see Section 2.2.1). If the non-configurable integrity verification had failed, for all policy levels, except `Hash` and `FileName`, WDAC blocks the execution of the image in `CipApplySIPolicyUMCI`. For the policy levels `Hash` and `FileName`, WDAC blocks, or allows, the execution of the image in `CipApplySIPolicyUMCI` only if WDAC verification fails, or succeeds, respectively. `CipApplySIPolicyUMCI` is not invoked when a critical system image is verified.

If a verified image is not digitally signed, WDAC performs image verification in an identical manner as in the scenario when the image is signed and the policy level `Hash` or `FileName` is configured. That is, WDAC blocks, or allows, the execution of the image in `CipApplySIPolicyUMCI` if WDAC verification fails, or succeeds, respectively. For unsigned images, only the policy levels `Hash` or `FileName` may be configured.

`SIPolicyValidateImage` verifies images based on data stored in an initialized WDAC policy (see Section 2.2.1) and brings the decision whether an image is allowed to execute. This section provides an overview of the working principles of `SIPolicyValidateImage`.

`SIPolicyValidateImage` verifies an image based on comparing:

- data stored in a deployed WDAC policy as part of file rules; with
- verification data associated with the image being verified. This work refers to this data as *image verification data*.

What image verification data is compared in `SIPolicyValidate` depends on the policy levels configured in the deployed WDAC policy. For example, this data involves the image's file name if the policy level `FileName` is configured (see Table 4).

`SIPolicyValidateImage` compares data stored in the deployed WDAC policy with image verification data using standard data comparison functions, such as memory and string comparison functions. Examples include `memcmp` [ms_memcmp] and `RtlEqualUnicodeString` [ms_str]. `SIPolicyValidateImage` accesses the content of the deployed WDAC policy through the `ci.dll` variable `g_SiPolicyHandles` (see Section 2.2.1.5). This variable stores at the offset `0x6C` the content of the policy in binary format (see Section 1.3). `g_SiPolicyHandles` is passed as the first parameter of `SIPolicyValidateImage`.

`g_SiPolicyHandles` is populated with the content of the deployed WDAC policy in the `SIPolicyInitialize` and `SIPolicySetActivePolicy` functions. These functions are invoked during the initialization of code integrity (see Section 2.2.1). Label *[1]* in Figure 23 depicts `SIPolicyInitialize` storing the content of a policy in `g_SiPolicyHandles`. Label *[2]* in Figure 23 depicts the policy content, stored in `g_SiPolicyHandles`, passed as a parameter to `SIPolicyValidateImage`. Label [3] in Figure 23 depicts the policy content as viewed with the `HxD` hex editor, in the context of the Windows 10 instance where the policy is deployed.

Image verification data is passed to `SIPolicyValidateImage` in the form of a structure, referred to as *validation context*. The validation context stores the image verification data that may be relevant when verifying an image based on any policy level. For example, the validation context stores:

- at offset `0x30` data related to the certificate chain used to sign the image being verified, including the leaf (i.e., the signer's certificate) and the PCAcertificate. This data is relevant when an image is verified based on the `Leaf` and `Publisher` policy levels (see Table 4);
- at offset `0x160` the name of the image being verified. This data is relevant when an image is verified based on the `FileName` policy level (see Table 4);

Figure 23: The content of a deployed DeviceGuard policy in different contexts

- at offset `0x1A0` the hash value of the image being verified. This data is relevant when an image is verified based on the `Hash` policy level (see Table 4).

The validation context is populated with data in multiple functions that are invoked before `SIPolicyValidateImage`. As an example, Figure 24 depicts Windows 10 populating the validation context with image verification data when the image `filecrypt.sys` is verified. This image is verified against a WDAC policy with the `PcaCertificate` policy level configured. The validation context is initialized in the `CipValidateImageHash` function, at the address `0xffffac05af19fbc0`. Once the validation context is initialized, the functions `CipCalculateImageHash`, `CipUpdateValidatationContextWithFileInfo`, and `MinCryptCopyPolicyInfo` populate the offsets `0x1A0`, `0x160`, and `0x30` of the validation context, respectively. `CipCalculateImageHash` calculates the image's hash value and stores it at the offset `0x1A0` of the validation context (see Figure 24, function `CipCalculateImageHash`). `CipUpdateValidatationContextWithFileInfo` extracts the name of the image from its properties and stores it at the offset `0x160` of the validation context (see Figure 24, function `CipUpdateValidatationContextWithFileInfo`).

`CipUpdateValidatationContextWithFileInfo` extracts the image's name and version from the image itself, by invoking the `SIPolicyGetOriginalFilenameAndVersionFromImage` function.

`filecrypt.sys` is signed through a catalog file (see Section 1.3). Therefore, the data related to the certificate chain used to sign the image is extracted from the Authenticode signature embedded in the catalog. This data is relevant for image verification, for example, when the `PcaCertificate` policy level is configured (see Table 4). The `I_FindFileOrHeaderHashInLoadedCatalogs` function searches through deployed catalog files for the image's hash value previously calculated by `CipCalculateImageHash`. In Windows 10, deployed catalog files are stored in the `%SystemRoot%\System32\CatRoot` folder. `I_FindFileOrHeaderHashInLoadedCatalogs` performs binary search in order to locate the image's hash value stored in a catalog (see `bsearch` [ms_bsrc] in Figure 24).

```
Breakpoint 0 hit
CI!CiValidateImageHeader:
fffff806`7675b400 4055            push    rbp
kd> !fileobj @rcx

\Windows\System32\drivers\filecrypt.sys
[...]
Breakpoint 1 hit
CI!CipValidateImageHash:
fffff806`7675cf2c 48895c2410      mov     qword ptr [rsp+10h],rbx
kd> ?? @rdx
unsigned int64 0xffffac05`af19fbc0
[...]
Breakpoint 3 hit
CI!CipCalculateImageHash:
fffff806`7675fa60 48895c2408      mov     qword ptr [rsp+8],rbx
kd> gu
CI!CipValidateFileHash+0x22c:
fffff806`7675d36c 8bf0            mov     esi,eax
kd> db 0xffffac05`af19fbc0+0x1A0
ffffac05`af19fd60  98 44 27 ed 77 d1 2f d4-ee 56 03 08 de e3 db 39  .D'.w./..V.....9
ffffac05`af19fd70  74 35 8a e2 00 00 00 00-00 00 00 00 00 00 00 00  t5..............
[...]
Breakpoint 2 hit
CI!CipUpdateValidatationContextWithFileInfo:
fffff806`76760044 4c8bdc          mov     r11,rsp
kd> gu
kd> db poi(0xffffac05`af19fbc0+0x160+0x8) L0x1a
ffffac05`af1f55d0  66 00 69 00 6c 00 65 00-63 00 72 00 79 00 70 00  f.i.l.e.c.r.y.p.
ffffac05`af1f55e0  74 00 2e 00 73 00 79 00-73 00                    t...s.y.s.
[...]
Breakpoint 4 hit
CI!I_FindFileOrHeaderHashInLoadedCatalogs:
fffff806`76755d68 48895c2408      mov     qword ptr [rsp+8],rbx
[...]
fffff806`76755e8b ff150794ffff    call    qword ptr [CI!_imp_bsearch (fffff806`7674f298)]
kd> t
nt!bsearch:
fffff803`f1dbf634 488bc4          mov     rax,rsp
kd> dps @rsp
ffffe581`18f05938  fffff806`76755e91 CI!I_FindFileOrHeaderHashInLoadedCatalogs+0x129
[...]
ffffe581`18f05960  fffff806`76757f10 CI!CipFileHashSearchCompareRoutineSHA1
[...]
Breakpoint 5 hit
CI!MincryptCopyPolicyInfo:
fffff806`7676b22c 488bc4          mov     rax,rsp
kd> gu
CI!I_FindFileOrHeaderHashInLoadedCatalogs+0x200:
fffff806`76755f68 8bd8            mov     ebx,eax
kd> db poi(0xffffac05`af19fbc0+0x30) L100
ffffac05`af541000  dc 11 00 00 00 00 00 00-88 10 54 af 05 ac ff ff  ..........T.....
[...]
ffffac05`af5410d0  30 82 01 22 30 0d 06 09-2a 86 48 86 f7 0d 01 01  0.."0.. *.H.....
ffffac05`af5410e0  01 05 00 03 82 01 0f 00-30 82 01 0a 02 82 01 01  ........0.......
[...]
Breakpoint 6 hit
nt!RtlDuplicateUnicodeString:
fffff803`f20cf3f0 488bc4          mov     rax,rsp
kd> du poi(@rdx+0x8)
ffffac05`af1e2708  "Microsoft-Windows-Desktop-Shared"
ffffac05`af1e2748  "-Drivers-onecore-Package~31bf385"
ffffac05`af1e2788  "6ad364e35~amd64~~10.0.14393.0.ca"
ffffac05`af1e27c8  "t."
[...]
```

Figure 24: Populating the validation context with image integrity verification data

Since `CipCalculateImageHash` has calculated a Secure Hash Algorithm (SHA)-1 hash value of the image, `bsearch` invokes the `CipFileHashSearchCompareRoutineSHA1` function in order to locate the SHA-1 hash in a catalog. The SHA-1 hash of `filecrypt.sys` is stored in the catalog `Microsoft-Windows-Desktop-Shared-Drivers-onecore-Package~31bf3856ad364e35~amd64~~10.0.14393.0.cat` (see Figure 24, function `RtlDuplicateUnicodeString`). Figure 25 depicts the SHA-1 hash of `filecrypt.sys` stored in this catalog as viewed with the `HxD` hex editor (see also Figure 24, function `CipCalculateImageHash`).

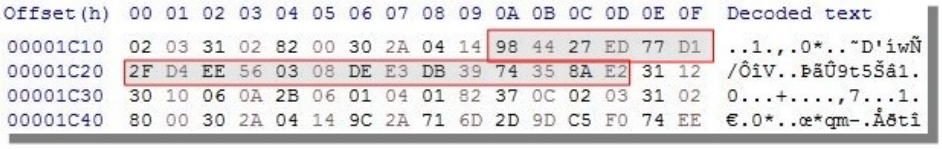

Figure 25: SHA-1 hash of filecrypt.sys stored in a catalog

Once `FindFileOrHeaderHashInLoadedCatalogs` has located the catalog, `MinCryptCopyPolicyInfo` stores data related to the certificate chain used to sign the catalog at the offset `0x30` of validation context (see Figure 24, function `MinCryptCopyPolicyInfo`). This includes the certificate chain itself. The binary sequence beginning with `30 82` depicted in Figure 24 mark certificate content encoded in the ASN.*1* format.

Once `MinCryptCopyPolicyInfo` has stored in the validation context data related to the certificate chain used to sign the catalog, `SIPolicyValidateImage` compares this data with data stored in the deployed WDAC policy (see Figure 26, label *[1],* function `memcmp`). The data `SIPolicyValidateImage` compares is a hash of the `TbsCertificate` field ([rfc_tbs], Section 4.1.1.1) of the PCAcertificate used to sign `Windows-Desktop-Shared-Drivers-onecore-Package~31bf3856ad364e35~amd64~~10.0.14393.0.cat` (`4e 80 be`... in Figure 26, label *[1]*). Figure 27 depicts the PCAcertificate that is extracted from Authenticode signature of the catalog. Figure 26, label *[2]*, depicts the extraction of the `TbsCertificate` field from the extracted PCAcertificate, depicted in Figure 27, with the `openssl` and `dd` utilities. It also depicts the calculated SHA-256 hash of the extracted `TbsCertificate` field with the `sha256sum` utility.

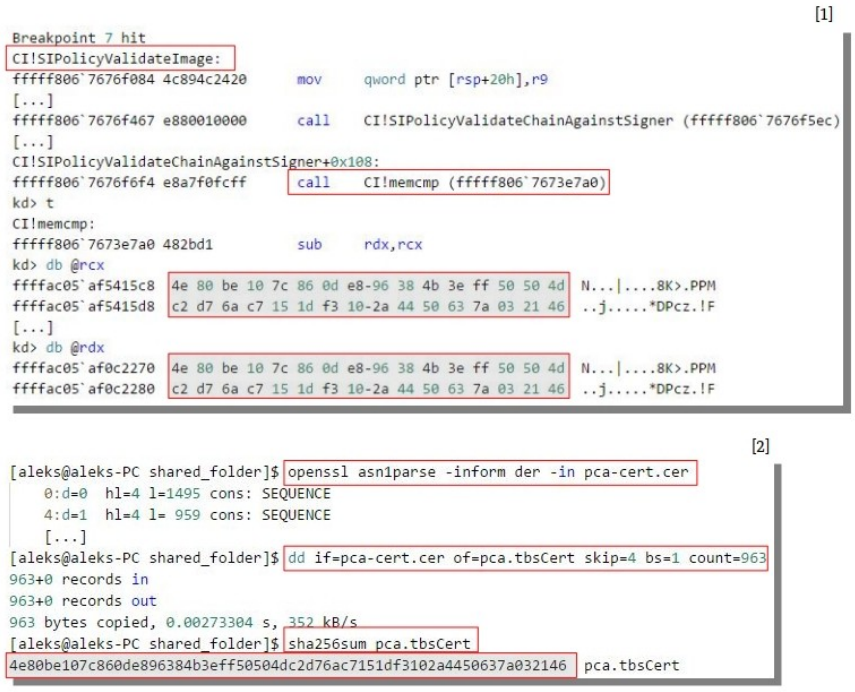

Figure 26: SIPolicyValidateImage comparing certificate data

Figure 26 shows that the data related to the certificate chain used to sign `filecrypt.sys`, compared in `SIPolicyValidateImage`, originates from the catalog that represents a detached signature of `filecrypt.sys`. In general, the discussion above shows that the image verification data compared in `SIPolicyValidateImage` originates either from the image being verified itself, or from the catalog serving as the image's detached signature.

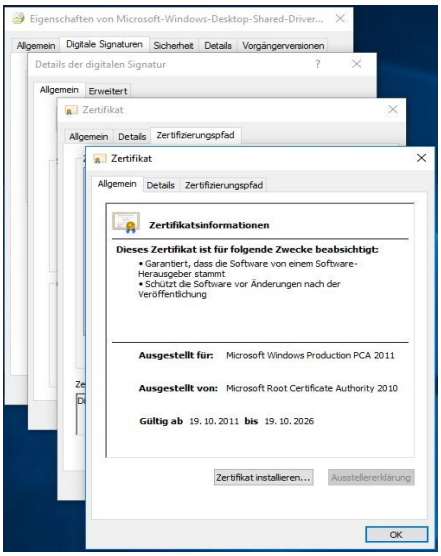

Figure 27: PCAcertificate extracted from a catalog

Once the validation context has been populated with data, it is passed to `SIPolicyValidateImage` for comparison with data originating from the deployed WDAC policy. The text blocks below provide an insight into the operation of `SIPolicyValidateImage`. They provide an overview of `SIPolicyValidateImage` comparing image verification data with data stored in a deployed WDAC policy when the policy levels `Hash`, `PcaCertificate`, and `Publisher` are configured (see also Figure 26). In the text blocks below:

- in the label *Policy and integrity verification:*
  - the field *Policy level* holds the configured policy level and the field *Policy generation* holds the command used to create the WDAC policy [ms_newci]. `$InitialCIPolicy` is a variable that specifies the path to the file in which the generated policy is to be stored;
  - the field *Verified data* holds the data based on which the image is verified (see Table 4);
  - the field *Verification function* holds the name of the function invoked by `SIPolicyValidateImage` in which data originating from the WDAC policy and image verification data is compared; and
  - the field *Comparison function(s)* holds the name of the function, or functions, that compare data originating from the WDAC policy with image verification data;
- in the label *Depiction* is placed a figure:
  - the label *[1]* of this figure depicts the execution of `SIPolicyValidateImage`, with a focus on what is documented in the fields *Verified data*, *Verification function*, and *Comparison function(s)*. This includes data stored in the validation context, data stored in the deployed WDAC policy, the function invoked by `SIPolicyValidateImage` in which this data is compared, and the function(s) with which the data is compared;
  - the label *[2]* of this figure depicts the data compared in `SIPolicyValidateImage`, extracted from the deployed WDAC policy in XML format with the `findstr` utility.

Policy and integrity verification

Policy level:
`Hash`
Policy generation:
`New-CIPolicy -Level Hash -FilePath $InitialCIPolicy –UserPEs`
Verified data:
the image's hash value (see Table 4)
Verification function:
`SIPolicyMatchFileRules`
Comparison function(s):
`memcmp` [ms_memcmp]

Depiction

[1]

```
Breakpoint 0 hit
CI!SIPolicyValidateImage:
[...]
kd> db poi ( poi(@rsp+0x38+0x8) + 0x8 ) L0x16
ffffb181`52dea060  64 00 75 00 6d 00 70 00-66 00 76 00 65 00 2e 00  d.u.m.p.f.v.e...
ffffb181`52dea070  73 00 79 00 73 00                              s.y.s.
[...]

CI!SIPolicyValidateImage+0x36c:
fffff804`d6cdf3f0 e853faffff      call    CI!SIPolicyMatchFileRules (fffff804`d6cdee48)
[...]
CI!SIPolicyMatchFileRules+0x1fc:
fffff804`d6cdf044 e857f7fcff      call    CI!memcmp (fffff804`d6cae7a0)
kd> t
CI!memcmp:
fffff804`d6cae7a0 482bd1          sub     rdx,rcx
kd> db @rcx
ffffb181`52d89eb0  21 17 ab 0b 78 6c 1e 89-5c 4e 28 33 0e 4a db 23  !...xl..\N(3.J.#
ffffb181`52d89ec0  d1 f5 de 41 71 ce 0f dd-f9 3c d3 1e 41 7d 91 a4  ...Aq....<..A}..
[...]
kd> db @rdx
ffffb181`528000ec  00 01 3a f8 35 30 1e 03-57 0c 85 20 67 fc 0f a0  ..:.50..W.. g...
ffffb181`528000fc  e7 17 65 94 67 f2 e2 ba-eb 39 30 47 ec 21 d7 54  ..e.g....90G.!.T
[...]
CI!memcmp:
fffff804`d6cae7a0 482bd1          sub     rdx,rcx
kd> db @rcx
ffffb181`52d89eb0  21 17 ab 0b 78 6c 1e 89-5c 4e 28 33 0e 4a db 23  !...xl..\N(3.J.#
ffffb181`52d89ec0  d1 f5 de 41 71 ce 0f dd-f9 3c d3 1e 41 7d 91 a4  ...Aq....<..A}..
[...]
kd> db @rdx
ffffb181`52800250  00 05 4c c3 27 30 2f 4e-35 2f d4 0d 7c 85 8c 02  ..L.'0/N5/..|...
ffffb181`52800260  7b 50 81 b1 7e 50 fd 39-71 73 11 8b da 4e 18 5e  {P..~P.9qs...N.^
[...]
```

[2]

```
C:\Users\ernw\Desktop>findstr /i "2117ab0b786c1e895c4e28330e4adb23" EnforcedPolicy.xml
    <Allow [...] FriendlyName="[...]dumpfve.sys Hash Sha256" Hash="2117AB0B786C1E895C4E28330E4ADB23
    D1F5DE4171CE0FDDF93CD31E417D91A4" />
    <Allow ID="ID_ALLOW_A_C946" FriendlyName="[...]dumpfve.sysdumpfve.sys Hash Sha256" Hash="2117AB0B786C1E895C4E28330E4ADB23
    D1F5DE4171CE0FDDF93CD31E417D91A4" />

C:\Users\ernw\Desktop>findstr /i "00013af835301e03570c852067fc0fa0" EnforcedPolicy.xml
    <Allow [...] FriendlyName="C:\Windows\[...]Activities.dll Hash Sha256" Hash="00013AF835301E03570C852067FC0FA0E717659
    467F2E2BAEB393047EC21D754" />
    <Allow [...] FriendlyName="C:\Windows\[...]Activities.dll Hash Sha256" Hash="00013AF835301E03570C852067FC0FA0E717659
    467F2E2BAEB393047EC21D754" />

C:\Users\ernw\Desktop>findstr /i "00054cc327302f4e352fd40d7c858c02" EnforcedPolicy.xml
    <Allow [...] FriendlyName="C:\Windows\[..]cemapi.dll.mui Hash Page Sha256" Hash="00054CC327302F4E352FD40D7C858C027B5081
    B17E50FD397173118BDA4E185E" />
    <Allow [...] FriendlyName="C:\Windows\[...]cemapi.dll.mui Hash Page Sha256" Hash="00054CC327302F4E352FD40D7C858C027B5081
    B17E50FD397173118BDA4E185E" />
```

Figure 28: Image verification: Policy level Hash

Policy and integrity verification

Policy level:
`PcaCertificate`
Policy generation:
`New-CIPolicy -Level PcaCertificate -FilePath $InitialCIPolicy –UserPEs`
Verified data:
the hash value of the `TcbCertificate` field of the PCAcertificate (see Table 4 and Figure 26)
Verification function:
`SIPolicyValidateChainAgainstSigner`
Comparison function(s):
memcmp [ms_memcmp]

Depiction

[1]

```
Breakpoint 0 hit
CI!SIPolicyValidateImage:
fffff806`1706f084 4c894c2420      mov     qword ptr [rsp+20h],r9
kd> db poi ( poi(@rsp+0x38+0x8) + 0x8 ) L0x18
ffff800d`79221b80  63 00 72 00 61 00 73 00-68 00 64 00 6d 00 70 00  c.r.a.s.h.d.m.p.
ffff800d`79221b90  2e 00 73 00 79 00 73 00                          ..s.y.s.
[...]
CI!SIPolicyValidateImage+0x3e3:
fffff806`1706f467 e880010000      call    CI!SIPolicyValidateChainAgainstSigner (fffff806`1706f5ec)
kd> t
[...]
CI!SIPolicyValidateChainAgainstSigner+0x108:
fffff806`1706f6f4 e8a7f0fcff      call    CI!memcmp (fffff806`1703e7a0)
kd> t
CI!memcmp:
fffff806`1703e7a0 482bd1          sub     rdx,rcx
kd> db @rcx
ffff800d`791a25c8  4e 80 be 10 7c 86 0d e8-96 38 4b 3e ff 50 50 4d  N...|....8K>.PPM
ffff800d`791a25d8  c2 d7 6a c7 15 1d f3 10-2a 44 50 63 7a 03 21 46  ..j.....*DPcz.!F
[...]
kd> db @rdx
ffff800d`790b588c  4e 80 be 10 7c 86 0d e8-96 38 4b 3e ff 50 50 4d  N...|....8K>.PPM
ffff800d`790b589c  c2 d7 6a c7 15 1d f3 10-2a 44 50 63 7a 03 21 46  ..j.....*DPcz.!F
[...]
```

[2]

```
C:\Users\ernw\Desktop>findstr /i "4e80be107c860de896384b3eff50504d" EnforcedPolicy_Pca.xml
    <CertRoot Type="TBS" Value="4E80BE107C860DE896384B3EFF50504DC2D76AC
    7151DF3102A4450637A032146" />
    <CertRoot Type="TBS" Value="4E80BE107C860DE896384B3EFF50504DC2D76AC
    7151DF3102A4450637A032146" />
```

Figure 29: Image verification: Policy level PcaCertificate

Policy and integrity verification

Policy level:
`Publisher`
Policy generation:
`New-CIPolicy -Level Publisher -FilePath $InitialCIPolicy –UserPEs`
Verified data:
the CN field of the certificate of the image's signer
the hash value of the `TcbCertificate` field of the PCAcertificate (see Table 4 and Figure 26)
Verification function:
`SIPolicyValidateChainAgainstSigner`
Comparison function(s):
`memcmp` [ms_memcmp], `RtlEqualUnicodeString` [ms_str]

Depiction

[1]

```
Breakpoint 0 hit
CI!SIPolicyValidateImage:
fffff808`0b21f084 4c894c2420      mov     qword ptr [rsp+20h],r9
kd> db poi ( poi(@rsp+0x38+0x8) + 0x8 ) L0x18
ffffd907`fcf172a0  73 00 74 00 6f 00 72 00-61 00 68 00 63 00 69 00  s.t.o.r.a.h.c.i.
ffffd907`fcf172b0  2e 00 73 00 79 00 73 00                          ..s.y.s.
[...]
CI!SIPolicyValidateImage+0x3e3:
fffff808`0b21f467 e880010000      call    CI!SIPolicyValidateChainAgainstSigner (fffff808`0b21f5ec)
[...]
CI!SIPolicyValidateChainAgainstSigner+0x108:
fffff808`0b21f6f4 e8a7f0fcff      call    CI!memcmp (fffff808`0b1ee7a0)
kd> t
CI!memcmp:
fffff808`0b1ee7a0 482bd1          sub     rdx,rcx
kd> db @rcx
ffffd907`fcde15c8  4e 80 be 10 7c 86 0d e8-96 38 4b 3e ff 50 50 4d  N...|....8K>.PPM
ffffd907`fcde15d8  c2 d7 6a c7 15 1d f3 10-2a 44 50 63 7a 03 21 46  ..j.....*DPcz.!F
[...]
kd> db @rdx
ffffd907`fccc6270  4e 80 be 10 7c 86 0d e8-96 38 4b 3e ff 50 50 4d  N...|....8K>.PPM
ffffd907`fccc6280  c2 d7 6a c7 15 1d f3 10-2a 44 50 63 7a 03 21 46  ..j.....*DPcz.!F
[...]
CI!SIPolicyValidateChainAgainstSigner+0x14a:
fffff808`0b21f736 e8abdffcff      call    CI!RtlEqualUnicodeString (fffff808`0b1ed6e6)
kd> t
CI!RtlEqualUnicodeString:
fffff808`0b1ed6e6 ff25141e0100    jmp     qword ptr [CI!_imp_RtlEqualUnicodeString (fffff808`0b1ff500)]
kd> t
nt!RtlEqualUnicodeString:
fffff800`75c72ae0 4883ec08        sub     rsp,8
kd> du poi(@rcx+0x8)
ffffd907`fcc94010  "Microsoft Windows"
kd> du poi(@rdx+0x8)
ffffd907`fccc62a0  "Microsoft Windows"
[...]
```

[2]

```
C:\Users\ernw\Desktop>findstr /i "4e80be107c860de896384b3eff50504d" EnforcedPolicy_Publisher.xml
    <CertRoot Type="TBS" Value="4E80BE107C860DE896384B3EFF50504DC2D76AC7151DF3102A4450637A032146" />
    <CertRoot Type="TBS" Value="4E80BE107C860DE896384B3EFF50504DC2D76AC7151DF3102A4450637A032146" />
    <CertRoot Type="TBS" Value="4E80BE107C860DE896384B3EFF50504DC2D76AC7151DF3102A4450637A032146" />
[...]

C:\Users\ernw\Desktop>findstr /i "CertPublisher.*Value.*Microsoft.*Windows.*" EnforcedPolicy_Publisher.xml
    <CertPublisher Value="Microsoft Windows" />
    <CertPublisher Value="Microsoft Windows" />
    <CertPublisher Value="Microsoft Windows 3rd party Component" />
    [...]
    <CertPublisher Value="Microsoft Windows Kits Publisher" />
    <CertPublisher Value="Microsoft Windows" />
```

Figure 30: Image verification: Policy level Publisher

3 Configuration and Logging Capabilities

This section provides an overview of the capabilities of Windows 10 for configuring WDAC (Section 3.1) and logging WDAC-related events (Section 3.2). Discussions about configuring the non-configurable code integrity functionalities of Windows 10 are not in the scope of this section. This is because they do not take user-defined criteria into account and do not expose significant configuration points to users (see Section 1.3).

3.1 Configuration Capabilities

This section provides an overview of the capabilities of Windows 10 for managing WDAC policies (Section 3.1.1). This involves policy creation, modification, and deployment. In addition, it discusses about what is important to be considered when WDAC is deployed at a given system (Section 3.1.2). Finally, it provides recommendations on constructing (Section 3.1.3) and creating WDAC policies (Section 3.1.4). The construction of a WDAC policy involves specifying policy rule options and policy levels associated with file rules (see Section 1.3).

3.1.1 WDAC Policy Management Capabilities

Windows 10 offers several mechanisms for managing a WDAC policy. For example, PowerShell ([ERNW WP2], Section 3.1.1) and the `System Center Configuration Manager` [ms_sccm] can be used for creating and modifying a WDAC policy. The System Center `Configuration Manager`, the `Group Policy Object Editor` utility [ms_gpo], and `Microsoft Intune` [ms_intune], can be used for deploying a WDAC policy. This section focuses on the mechanisms for managing WDAC policies that are distributed with Windows 10 – PowerShell (Section 3.1.1.1) and the `Group Policy Object Editor` utility (Section 3.1.1.2). This excludes mechanisms such as `Microsoft Intune` and the `System Center Configuration Manager`.

3.1.1.1 PowerShell

WDAC policies can be created and modified by executing PowerShell cmdlets for WDAC management. These cmdlets are implemented as part of the PowerShell module `ConfigCI`. A comprehensive description of the cmdlets for managing WDAC policies is available at [ms_pscci]. In this section, we provide summarizing descriptions of some of these cmdlets, whose use is crucial for managing WDAC policies in practice.

New-CIPolicy This cmdlet creates a new WDAC policy in XML format (see Section 1.3):

- the `ScanPath` parameter of `New-CIPolicy` specifies a filesystem path. The cmdlet scans this path in-depth for images. It also automatically generates file rules (see Section 1.3) for each image that is stored at the path. These file rules are then stored in the WDAC policy in XML format.

  The `New-CIPolicy` cmdlet does not verify the validity of the specified filesystem path with the `ScanPath` parameter. It generates a WDAC policy that does not contain file rules if an invalid filesystem path is provided. If the `ScanPath` parameter is not used, `New-CIPolicy` scans the disk volume where Windows is installed.

  If the `ScanPath` parameter is used when creating a new WDAC policy, the parameter should specify a filesystem path at which files whose integrity is important to be verified by WDAC are stored. The user creating a WDAC policy brings the decision on what path is specified based on system security requirements. However, it is important to emphasize that once a WDAC policy is deployed, Windows 10 performs WDAC verification for all images that it loads (see Section 2.2.2). This implies that the deployed policy should contain file rules that allow the execution of all images that are needed for Windows 10 to

operate. This includes the kernel and driver images. If that is not the case, Windows 10 may not start up due to failed WDAC verification of required system images.

- the `Audit` parameter of `New-CIPolicy` configures `New-CIPolicy` to scan for driver images indicated in the Windows log that stores WDAC events logged by a previously deployed WDAC policy (see Section 3.2). The cmdlet then automatically generates file rules for each image indicated in the log file. These file rules are then stored in the WDAC policy in XML format;
- the `Level` parameter of `New-CIPolicy` specifies a policy level. `New-CIPolicy` associates this policy level with the file rules specified as part of the WDAC policy being generated (see Table 4);
- the `Fallback` parameter of `New-CIPolicy` specifies a single or multiple policy levels that may be associated with file rules specified as part of the new WDAC policy (see Table 4). This takes place only if the policy level specified by the `Level` parameter cannot be associated with a given file rule. For example, the `Publisher` policy level cannot be associated with the file rule for verifying an image that is not digitally signed. If the `Fallback` parameter specifies the `Hash` policy level, this level will be associated with the file rule. The policy levels specified by the `Fallback` parameter are referred to as fallback policy levels in this work;
- the `UserPEs` parameter sets the `Enabled:` UMCI policy rule option (see Table 3) in the WDAC policy being generated;
- the `Deny` parameter specifies that `New-CIPolicy` will generate only deny file rules; that is, the execution of images specified as part of these rules is blocked. If this parameter is not specified, `New-CIPolicy` generates allow file rules; that is, the execution of images specified as part of these rules is allowed;
- the `FilePath` parameter of `New-CIPolicy` specifies a filesystem path at which the newly generated WDAC policy in XML format is to be stored.

More detailed information on the `New-CIPolicy` cmdlet, including usage examples, can be found at [ms_newci].

Merge-CIPolicy This cmdlet merges several WDAC policies in XML format into a single WDAC policy:
- the `PolicyPaths` parameter of `Merge-CIPolicy` specifies filesystem paths at which the WDAC policies to be merged are stored;
- the `OutputFilePath` parameter of `Merge-CIPolicy` specifies a filesystem path at which the newly generated WDAC policy is to be stored.

More detailed information on the `Merge-CIPolicy` cmdlet, including usage examples, can be found at [ms_mrg].

ConvertFrom-CIPolicy This cmdlet converts a WDAC policy in XML format into binary format, ready for deployment (see Section 1.3):
- the `XmlFilePath` parameter of `ConvertFrom-CIPolicy` specifies a filesystem path at which a WDAC policy in XML format is stored;
- the `BinaryFilePath` parameter of `ConvertFrom-CIPolicy` specifies a filesystem path at which the converted WDAC policy in binary format is to be stored.

More detailed information on the `ConvertFrom-CIPolicy` cmdlet, including usage examples, can be found at [ms_conv].

Set-RuleOption This cmdlet sets or unsets (i.e., deletes) a policy level option (see Table 3) as part of a WDAC policy in XML format:
- the `FilePath` parameter of `Set-RuleOption` specifies a filesystem path at which the WDAC policy in XML format is stored;

- the `Help` parameter of `Set-RuleOption` displays all available policy level options, which are associated with unique index numbers;
- the `Option` parameter specifies an index number associated with a given policy level option. This indicates the policy level option to be set, or unset, as part of the WDAC policy;
- the `Delete` parameter of `Set-RuleOption` unsets the policy level option associated with the index number specified by the `Option` parameter. The `Delete` parameter is optional; if not specified, `Set-RuleOption` sets the policy level option associated with the index number specified by `Option`.

More detailed information on the `Set-RuleOption` cmdlet, including usage examples, can be found at [ms_sro].

Add-SignerRule This cmdlet modifies an existing WDAC policy in XML format such that it generates a file rule. This rule allows or denies the execution of the entities operating in user- and/or kernel-mode that are specified in the WDAC policy and are signed by a user-specified certificate. This also includes the WDAC policy itself, if signed. `Add-SignerRule` then adds the generated rule to the WDAC policy:

- the `FilePath` parameter of `Add-SignerRule` specifies a filesystem path at which the WDAC policy in XML format is stored;
- the `CertificatePath` parameter of `Add-SignerRule` specifies a filesystem path at which the certificate used to sign entities operating in user- or kernel-mode is stored;
- the `User` and `Kernel` parameters configure `Add-SignerRule` to generate a file rule, which allows or blocks the execution of the entities operating in user- and kernel-mode, which are specified in the WDAC policy;
- the `Deny` parameter configures `Add-SignerRule` to generate a deny file rule. If this parameter is not specified, `Add-SignerRule` generates an allow file rule (see paragraph *New-CIPolicy*).

More detailed information on the `Add-SignerRule` cmdlet, including usage examples, can be found at [ms_addsr].

3.1.1.2 Group Policy Object Editor

The `Group Policy Object Editor` utility can be used for deploying a WDAC policy. The group policy setting for deploying a WDAC policy is located at the policy path `Computer Configuration\Administrative Templates\System\Device Guard\Deploy Code Integrity Policy` (see Figure 31). The field `Code Integrity Policy file path` accepts a path to a WDAC policy in binary format. This WDAC policy is deployed after system reboot (see Section 1.3, Figure 4).

3.1.2 WDAC Deployment Considerations

The effective deployment of WDAC implies:

- ensuring an actual security benefit: This implies firstly a correct construction of a WDAC policy. This, in turn, involves constructing a WDAC policy that takes into account the malicious scenarios that are relevant to the system on which the policy is deployed;
- ensuring manageable operation and security of the WDAC policy: This implies the design and realization of a WDAC deployment scenario that takes into account deployment specialties as well as security measures.

A set of relevant, interrelated WDAC deployment considerations enable the security-efficient construction of a WDAC policy, and the design and realization of WDAC deployment scenarios. These considerations can be summarized as follows:

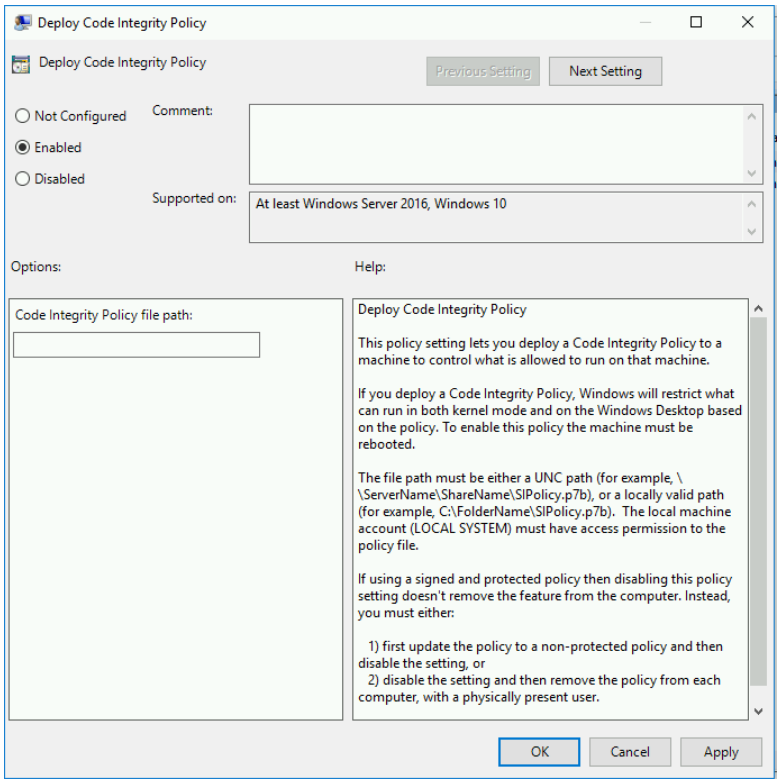

Figure 31: Deployment of a WDAC *policy: Group Policy Object Editor*

- *definition of the scope and the goal of* WDAC *deployment*: WDAC provides security benefits only in specific scenarios. For example, WDAC enables the control over the execution of images at system-wide level (see Section 1.3). However, if image execution control at user-level, or user group-level, is needed, alternative mechanisms should be deployed in addition to WDAC. An example is Microsoft AppLocker, which enables image execution control at user-level [ms_appl];
  - *identification of systems at which the deployment of WDAC is not applicable*: Not all systems can be protected by WDAC in an effective and straightforward manner. Such systems are, for example, systems used for code development. This is because they execute newly created and unknown code on a regular basis. Another example are systems with many software installation changes over time, such as systems for software testing. The correct construction of a WDAC policy for deployment on such a system may be a challenging and error-prone task. In such a scenario, the design and deployment of alternative mechanisms enabling image execution control may be considered;
- *blocking the execution of images that allow the arbitrary execution of code*: WDAC is ineffective when it comes to the blocking of execution of arbitrary code by an image that is allowed to execute by a deployed WDAC policy. This execution effectively bypasses WDAC protections. An example image that executes arbitrary code is `windbg`. A correctly constructed WDAC policy should not allow the execution of images that may execute arbitrary code. [ms_arb] provides an updated list of such images.

  When WDAC is deployed on a given system, there should be an ongoing process for the identification of images that may execute arbitrary code. For example, this process may involve the identification of images that have command-line arguments that invoke other images, or of images that invoke functions that execute arbitrary code, such as `Assembly.Load` [ms_al]. Scripts, such as PowerShell scripts, may also execute arbitrary code. [ERNW WP8] provides recommendations on preventing such scripts from executing code. An offensive attempt to bypass the deployed WDAC policy through the execution of arbitrary code may also be considered as an approach to evaluate WDAC deployment;
- *management of the signing certificate and process*: Any WDAC policy that is to be deployed should be digitally signed with a code signing certificate in order to prevent unauthorized policy modifications (see

Section 1.3). In organizations, this may require a proper managed PKI. The process itself, of signing a WDAC policy should be conducted on a dedicated, security-hardened system so that the signing certificate and process are secure;

- *deployment of monitoring and analysis mechanisms*: The deployment of centralized monitoring and analysis mechanisms should be considered. These mechanisms should monitor and analyze logged events by the deployed WDAC policy, such as policy violations. In the case of policy violations, appropriate follow-up procedures should be enforced;
- *implementation of an image lifecycle policy*: Since WDAC controls what images are allowed to execute on a given system, the implementation of an image lifecycle policy should be considered. This policy should track any changes in the landscape of images that are deployed on the system, such as newly deployed images, or versions of images. It should also reflect these changes in the deployed WDAC policy. For example, the image lifecycle policy should involve processes that ensure that potentially malicious images, such as outdated images, or images that allow the execution of arbitrary code, are specified in the deployed WDAC policy. In addition, the image lifecycle policy should involve processes that ensure that newly deployed images are added to the WDAC policy. [ERNW WP9] provides a lifecycle policy for images of Universal Windows Apps ([ERNW WP2], Section 3.6).

3.1.3 Recommendations on Constructing WDAC Policies

3.1.3.1 Recommendations on Policy Rule Options

Table 6 provides recommendations for setting (*set* in Table 6) or unsetting (*unset* in Table 6) each of the policy rule options listed in Table 3 (column 'Recommendation' in Table 6). The 'Comments' column of Table 6 provides the rationale behind each recommendation. It also provides an overview of potential challenges for implementing the recommendation, and the scenarios in which the recommendation is practically useful. Most of the challenges discussed in Table 6 are related to the trade-off between security and potential practical implications of setting or unsetting a given policy rule option.

Policy rule option	Recommendation	Comments
`Enabled:` **`UMCI`**	set	This option applies the deployed WDAC policy to entities that operate in user- and in kernel-mode. By default, a WDAC policy applies only to entities that operate in kernel-mode. Entities that operate in user-mode may compromise the confidentiality, integrity, and/or the availability of a system. Therefore, this policy rule option should be set. The creation and on-going management of WDAC policies that include entities that operate in user-mode is a challenging task. For example, if the landscape of images that are deployed on a given system is changing frequently, the deployed WDAC policy needs to be also frequently updated. Therefore, the practical feasibility of setting this policy rule option needs to be evaluated. `Enabled:` UMCI is set if the `-UserPEs` parameter of the `New-CIPolicy` PowerShell cmdlet is specified [ms_newci].
`Enabled: Boot Menu Protection`	unset	Currently not supported.

Policy rule option	Recommendation	Comments
`Required: WHQL`	set	This policy rule option should be set only if all drivers used by the system on which a WDAC policy is deployed have passed the WHCK tests and are signed by WHQL [ms_whqlsig] [ms_whlk]. This prevents any other driver from executing. This includes, for example, sophisticated malware signed by certificates specified as trusted in the WDAC policy.
`Enabled: Audit Mode`	set during testing, unset otherwise	This policy rule option should be set only when testing a newly created or modified WDAC policy. Setting this option helps to evaluate the effects of such a policy. `Enabled: Audit Mode` should be unset when a WDAC policy is enforced. This policy rule option is set by default when a WDAC policy is created with the `New-CIPolicy` PowerShell cmdlet (see Section 3.1.1.1).
`Disabled: Flight Signing`	unset	This policy rule option should be unset if no flight signed images run on the system at which a WDAC policy is deployed. Blocking the execution of flight signed images improves system security. This is because such images may implement not thoroughly tested, new features that might contain critical vulnerabilities. In addition, certificates used for signing flight signed images may be exposed to a bigger group of human actors with less security restrictions when compared with certificates used for signing production-ready images. Therefore, the risk of these certificates being compromised is higher. In such a scenario, a compromised certificate used for flight signing can be used to sign malicious images.
`Enabled: Inherit Default Policy`	unset	Currently not supported.
`Enabled: Unsigned System Integrity Policy`	unset	This policy rule option should be unset such that only digitally signed WDAC policies may be deployed (see Section 1.3). If a proper process for managing the certificate used to sign a given WDAC policy is conducted (see Section 3.1.2), `Enabled: Unsigned System Integrity Policy` prevents a malicious actor from tampering with the WDAC policy, for example, to allow malicious images to be executed. This policy rule option is set by default when a WDAC policy is created with the `New-CIPolicy` PowerShell cmdlet (see Section 3.1.1.1).
`Allowed: Debug Policy Augmented`	unset	Currently not supported.
`Required: EV Signers`	set	This policy rule option should be set only if all drivers used by the system on which a WDAC policy is deployed are signed by EV certificates. For an EV certificate to be issued to a given entity, the entity is subjected to a vetting by a

Policy rule option	Recommendation	Comments
		certificate authority. This improves system security, since EV certificates require additional efforts by malicious actors to be compromised.
`Enabled: Advanced Boot Options Menu`	set during testing, unset otherwise	This option should be set only when testing a newly created or modified WDAC policy. When a WDAC policy is not being tested, the option prevents a malicious actor from potentially bypassing WDAC protections by exploiting currently unknown WDAC bypass venues. This policy rule option is set by default when a WDAC policy is created with the `New-CIPolicy` PowerShell cmdlet (see Section 3.1.1.1).
`Enabled: Boot Audit on Failure`	set during testing, unset otherwise	This policy option configures a WDAC policy, operating in enforcement mode, to switch to audit mode if image verification fails during system startup. This option should be set only when testing a newly created or modified WDAC policy. When a WDAC policy is not being tested, system boot should fail if image verification fails; that is, `Enabled: Boot Audit on Failure` should be unset. `Enabled: Boot Audit on Failure` may be set in the scenario where the goal is to monitor attacker activities for attack analysis purposes. However, this poses the risk of the attacker compromising the targeted system.
`Disabled: Script Enforcement`	unset	Currently not supported.
`Required: Enforce Store Applications`	set	This policy level option should be set for added system security. This policy rule option is set by default when a WDAC policy is created with the `New-CIPolicy` PowerShell cmdlet (see Section 3.1.1.1).
`Enabled: Managed Installer`	set	This policy level option should be set if images installed by a software distribution solution, such as the System Center Configuration Manager [ms_sscm], are deployed on a given system.
`Enabled: Intelligent Security Graph Authorization`	unset for static image landscapes, set for dynamic image landscapes	This option automatically allows for any image classified as "known good" by the `Intelligent Security Graph` [ms_inteli] to execute. If the landscape of images, deployed on a given system and considered trusted, is static (i.e., it does not frequently change over time), this policy option should be unset. This is to prevent the potential execution of images that are new to the system's image landscape and that have been falsely labeled as "known good" by `Intelligent Security Graph`. In addition, `Intelligent Security Graph`

Policy rule option	Recommendation	Comments
		may transmit data over the Internet in order to evaluate (i.e., classify) an image. This may have an impact on users' privacy in terms of what images they execute. If the landscape of images, deployed on a given system and considered trusted, is dynamic (i.e., it frequently changes over time), this policy option should be set in order to reduce WDAC policy management efforts (see Section 3.1.2). However, the risk of executing images falsely labeled as "known good" by `Intelligent Security Graph` should be taken into account.
`Enabled: Invalidate EAs on Reboot`	set, if `Enabled: Intelligent Security Graph Authorization` is set	Since the evaluation of images by the `Intelligent Security Graph` may change over time, they should be re-evaluated at system reboot. Therefore, this option should be set, if `Enabled: Intelligent Security Graph Authorization` is set.
`Enabled: Update Policy No Reboot`	set	This option allows for modifications to an already deployed WDAC policy to be applied without system reboot. By default, for changes to a deployed WDAC policy to take effect, the system at which the policy is deployed has to be rebooted. Therefore, if a given system is not rebooted frequently, the application of any changes to the deployed WDAC policy takes time. In a scenario where immediate policy changes are critical for system security, this policy option should be set.

Table 6: Recommendations on policy rule options

3.1.3.2 Recommendations on Policy Levels

Table 7 provides recommendations on the usage of the policy levels listed in Table 4 (column 'Recommendation' in Table 7).

Policy level	Recommendation
`Hash`	This policy level is the most fine-grained level since it verifies individual images based on their unique hash values. It requires a deployed WDAC policy to be updated whenever any image specified in the policy is modified. This may be an operationally challenging task. Therefore, the `Hash` policy level should be used only for protecting highly security-critical systems, or in scenarios where the use of the other policy levels is not applicable.
`FileName`	The use of this policy level is generally not recommended. The name of an image is not a unique image attribute since it may be changed, or assigned to another image with different contents. This implies that replacing a legitimate image with a malicious image with the same name as the legitimate image would be allowed to execute by WDAC. This scenario does not hold for critical system images (e.g., the Windows kernel, see Section 2.2.2).
`LeafCertificate`	This policy level should be used for the verification of digitally signed

Policy level	Recommendation
	images that can be associated with a specific certificate issued to the image's signer. This certificate is the leaf of the certificate chain used to sign the image (see Table 4). The content of the image can then be changed without the need to update a deployed WDAC policy as long as the image's signing certificate is not changed. At the one hand, this policy level enables a fine-grained control over what images are allowed execute based on specific signing certificates. However, a deployed WDAC policy enforcing this level has to be updated in case these certificates are changed, for example, due to certificate revocations.
`PcaCertificate`	This policy level is a coarse-grained level since it verifies images based on their PCAcertificates. This implies that all images that share a single PCAcertificate may be allowed to execute if this certificate is considered trusted. Although it enforces a less strict control than many other policy levels (e.g., `LeafCertificate`), `PcaCertificate` makes the management of WDAC policies easier. This policy level should be chosen for use over `LeafCertificate` only if the signing certificates of the images deployed on a given system are changed frequently. In such a scenario, updating a deployed WDAC policy is an operationally challenging task.
`RootCertificate`	Currently not supported.
`Publisher`	This policy level refines the `PcaCertificate` level, enabling a more strict control based on the CN field of the leaf certificate in the certificate chain used to sign a given image. However, a deployed WDAC policy needs to be changed if this field is changed. `Publisher` should be chosen for use over `PcaCertificate` if updating a deployed WDAC policy when required is an operationally feasible task.
`SignedVersion`	This policy level refines the `Publisher` level, enabling a more strict control based on the file version of a given image. However, a deployed WDAC policy needs to be changed if this version is changed. `SignedVersion` should be chosen for use over `Publisher` if updating a deployed WDAC policy when required is an operationally feasible task.
`FilePublisher`	This policy level refines the `SignedVersion` level, enabling a more strict control based on the name of a given image. However, a deployed WDAC policy needs to be changed if this name is changed. `FilePublisher` should be chosen for use over `SignedVersion` if updating a deployed WDAC policy when required is an operationally feasible task.
`WHQL`	This policy level is a coarse-grained level since it allows any image signed by the WHQL to execute. This policy level should be used if images signed by the WHQL, primarily driver images, are deployed on a given system protected by WDAC.

Policy level	Recommendation
`WHQLPublisher`	This policy level refines the `WHQL` level, enabling a more strict control based on the CN field of the leaf certificate in the certificate chain used to sign a given image. However, a deployed WDAC policy needs to be changed if this field is changed. `WHQLPublisher` should be chosen for use over `WHQL` if updating a deployed WDAC policy when required is an operationally feasible task.
`WHQLFilePublisher`	This policy level refines the `WHQLPublisher` level, enabling a more strict control based on the file version of a given image. However, a deployed WDAC policy needs to be changed if this version is changed. `WHQLFilePublisher` should be chosen for use over `WHQLPublisher` if updating a deployed WDAC policy when required is an operationally feasible task.

Table 7: Recommendations on policy levels

3.1.4 Recommendations on Creating WDAC Policies

This section discusses recommended approaches for creating a WDAC policy (paragraph 'Approach #1', ' Approach #2', and ' Approach #3') commonly seen in practice. It also provides a summarized overview of the advantages and disadvantages of each approach.

Approach #1 A security administrator creates a WDAC policy in XML format that contains only allow file rules for the verification of images that are deployed on a "golden" system (see Section 3.1.1.1, paragraph 'New-CIPolicy') . A "golden" system is a system for which it is assumed that the deployed images on this system are trusted. A WDAC policy creation on a "golden" system is a straightforward task. However, due to the potentially big number of images that may be deployed on the system, the security administrator conducts a careful review of the created WDAC policy. This is to ensure that the policy does not contain any file rules that allow the execution of images whose execution should be blocked. In addition to the policy review, the security administrator creates a separate WDAC policy that contains only deny file rules. These file rules are for the verification of images that allow the arbitrary execution of code (see Section 3.1.2, [ms_arb]). The administrator then merges this policy with the WDAC policy created on the "golden" system (see Section 3.1.1.1, paragraph 'Merge-CIPolicy'). The merged policy is then converted into binary format (see Section 3.1.1.1, paragraph 'ConvertFrom-CIPolicy') and deployed where needed. The security administrator first tests the policy by configuring it to operate in audit mode, and then configures the policy to operate in enforced mode (see Section 3.1.1.1, paragraph 'Set-RuleOption').

Approach #2 A security administrator creates a WDAC policy in XML format that contains deny rules for all images deployed on a system (see Section 3.1.1.1, paragraph 'New-CIPolicy') . The administrator then modifies the policy such that only images that are signed by Microsoft are allowed to execute [ms_pmix]. In addition, the administrator carefully reviews and modifies the policy, such that trusted images, whose execution has been blocked, are now allowed. Such policy modifications are typically done manually by editing the WDAC policy in XML format. This approach has the advantage of not manually populating a WDAC policy with allow rules for images signed by Microsoft. This is because such images are common on Windows systems. To this end, the `New-CIPolicy` cmdlet can be used to automatically generate allow rules for these images (see Section 3.1.1.1, paragraph 'New-CIPolicy'). However, some images signed by Microsoft may allow the arbitrary execution of code [ms_arb]. The security administrator creates a separate WDAC policy that contains only deny file rules for the verification of these images. The administrator then merges this policy with the previously created policy. The merged WDAC policy is then converted into binary format (see Section 3.1.1.1, paragraph 'ConvertFrom-CIPolicy') and deployed on the system. The security administrator first tests the policy by configuring it to operate in audit mode, and then configures the policy to operate in enforced mode (see Section 3.1.1.1, paragraph 'Set-RuleOption').

Approach #3 This approach is similar to Approach #2, with the difference of resulting in a more strict WDAC policy. A security administrator creates a WDAC policy in XML format that contains deny rules for all images deployed on a system (see Section 3.1.1.1, paragraph 'New-CIPolicy'). The administrator then carefully reviews and modifies the policy, such that trusted images, whose execution has been blocked, is now allowed. The security administrator also creates a separate WDAC policy that contains only deny file rules for the verification of images that allow the arbitrary execution of code [ms_arb]. The administrator then merges this policy with the previously created WDAC policy. The merged policy is then converted into binary format (see Section 3.1.1.1, paragraph 'ConvertFrom-CIPolicy') and is deployed on the system. The security administrator first tests the policy by configuring it to operate in audit mode, and then configures the policy to operate in enforced mode (see Section 3.1.1.1, paragraph 'Set-RuleOption').

The above approaches for creating WDAC policies are best applicable in different scenarios, depending on the system where the policies are deployed. This section discusses next two example scenarios – 'Workstation' and 'Server'.

Scenario: Workstation

This scenario assumes a system with a dynamic, frequently changing image landscape. Approach #2 is an optimal approach for creating a WDAC policy to be deployed on such a system. This is because it strikes a good balance between the strictness and the flexibility of the created WDAC policy. This balance is achieved such that the WDAC policy is created to contain file rules associated with the policy level `Publisher`, with fallback policy levels `PcaCertificate` and `Hash` (see Section 3.1.1.1, paragraph 'New-CIPolicy'). In addition, the following policy rule options are recommended to be set at policy creation: `Enabled: UMCI`, `Required: WHQL`, `Disabled: Flight Signing`, and `Required: Enforce Store Applications` (see Table 6).

Scenario: Server

This scenario assumes a security-critical system with a relatively static image landscape. Approach #3 is an optimal approach for creating a WDAC policy to be deployed on such a system. This is because it results in a very strict WDAC policy, which is appropriate for security-critical systems. This strictness is achieved such that the WDAC policy is created to contain file rules associated with the fine-grained policy level `FilePublisher` (see Table 7), with fallback policy levels `WHQLFilePublisher` and `Hash` (see Section 3.1.1.1, paragraph 'New-CIPolicy'). In addition, the following policy rule options are recommended to be set at policy creation: `Enabled: UMCI`, `Required: WHQL`, `Disabled: Flight Signing`, `Required: EV Signers`, and `Required: Enforce Store Applications` (see Table 6). For highly security-critical systems with a static image landscape, a `WDAC` policy may be created such that it contains file rules associated with the most fine-grained policy level – `Hash`, with no fallback levels.

3.2 Logging Capabilities

Windows 10 uses the Event Tracing for Windows (ETW) framework ([ERNW WP2], Section 4.1) for logging WDAC events and events produced by the non-configurable code integrity functionalities. Table 8 provides the names and the globally unique identifiers (GUIDs) of relevant ETW providers. The `Microsoft-Windows-Codeintegrity` and the `Microsoft-Windows-AppLocker` providers log events related to objects, such as applications, executables or scripts, being blocked with respect to a deployed WDAC policy. The `Microsoft-Windows-Codeintegrity` provider also logs events produced by the non-configurable code integrity functionalities of Windows 10. The `Microsoft-Windows-DeviceGuard` provider logs events related to the processing of a deployed WDAC policy [ms_log].

ETW provider	GUID
`Microsoft-Windows-Applocker`	`CBDA4DBF-8D5D-4F69-9578-BE14AA540D22`
`Microsoft-Windows-CodeIntegrity`	`4EE76BD8-3CF4-44A0-A0AC-3937643E37A3`
`Microsoft-Windows-DeviceGuard`	`F717D024-F5B4-4F03-9AB9-331B2DC38FFB`

Table 8: ETW *providers logging code integrity-related events*

The tables in the Appendix, section 'Event IDs', present the Event IDs, and their descriptions, under which the `Microsoft-Windows-CodeIntegrity`, `Microsoft-Windows-Applocker`, and `Microsoft-Windows-DeviceGuard` providers may log events (column 'Event ID' and 'Event Message' respectively). The `wevtutil` utility displays the Event IDs and their descriptions ([ERNW WP2], Section 4.3). In the tables in the Appendix, section 'Event IDs', numbers proceeded by the percent sign (%) mark dynamic content generated at runtime. The event descriptions in these tables are as provided by Microsoft. The `Microsoft-Windows-Applocker` provider may log additional events than those listed in the Appendix. However, these events are not related to WDAC and therefore, are not listed in the Appendix.

Appendix

Tools

Tool	Availability and description
IDA	*Availability:* https://www.hex-rays.com/products/ida/index.shtml [Retrieved: 12/9/2018] *Description*: A disassembly and debugging framework.
Windows Debugger (windbg)	*Availability:* https://developer.microsoft.com/en-us/windows/hardware/download-windbg [Retrieved: 12/9/2018] *Description*: A debugger for the Windows system.
Group Policy Object Editor	*Availability:* Distributed with Windows 10 *Description*: A tool for configuring group policies.
wevtutil	*Availability:* Distributed with Windows 10 *Description*: A tool for querying running logging mechanisms.

A WDAC Policy

```
<?xml version="1.0" encoding="utf-8"?>
<SiPolicy xmlns="urn:schemas-microsoft-com:sipolicy">
  <VersionEx>10.0.0.0</VersionEx>
  <PolicyTypeID>{A244370E-44C9-4C06-B551-F6016E563076}</PolicyTypeID>
  <PlatformID>{2E07F7E4-194C-4D20-B7C9-6F44A6C5A234}</PlatformID>
  <Rules>
    <Rule>
      <Option>Enabled:Unsigned System Integrity Policy</Option>
    </Rule>
    <Rule>
      <Option>Enabled:Advanced Boot Options Menu</Option>
    </Rule>
   [...]
  </Rules>
  <!--EKUS-->
  <EKUs />
  <!--File Rules-->
  <FileRules />
  <!--Signers-->
  <Signers>
    <Signer ID="ID_SIGNER_S_1" Name="Microsoft Windows Production PCA 2011">
      <CertRoot Type="TBS"
Value="4E80BE107C860DE896384B3EFF50504DC2D76AC7151DF3102A4450637A032146" />
    </Signer>
    <Signer ID="ID_SIGNER_S_12" Name="Microsoft Code Signing PCA">
      <CertRoot Type="TBS" Value="27543A3F7612DE2261C7228321722402F63A07DE" />
    </Signer>
    [...]
  </Signers>
  <!--Driver Signing Scenarios-->
  <SigningScenarios>
    <SigningScenario Value="131" ID="ID_SIGNINGSCENARIO_DRIVERS_1" FriendlyName="Auto generated
policy on 07-10-2018">
      <ProductSigners>
        <AllowedSigners>
          <AllowedSigner SignerId="ID_SIGNER_S_20" />
```

```
          <AllowedSigner SignerId="ID_SIGNER_S_21" />
          [...]
        </AllowedSigners>
      </ProductSigners>
    </SigningScenario>
    <SigningScenario Value="12" ID="ID_SIGNINGSCENARIO_WINDOWS" FriendlyName="Auto generated policy
on 07-10-2018">
      <ProductSigners>
        <AllowedSigners>
          <AllowedSigner SignerId="ID_SIGNER_S_1" />
          <AllowedSigner SignerId="ID_SIGNER_S_12" />
          [...]
        </AllowedSigners>
      </ProductSigners>
    </SigningScenario>
  </SigningScenarios>
  <UpdatePolicySigners />
  <CiSigners>
    <CiSigner SignerId="ID_SIGNER_S_1" />
    <CiSigner SignerId="ID_SIGNER_S_12" />
    [...]
  </CiSigners>
  <HvciOptions>0</HvciOptions>
</SiPolicy>
```

Functionalities Exported by ci.dll

Function	Category					
	Image integrity verification				Initialization	Miscellaneous
	Image sections		Signature placement			
	Page	File	Embedded	Catalogs		
CiCheckSignedFile		*	*			
CiFindPageHashesInCatalog	*			*		
CiFindPageHashesInSignedFile	*		*			
CiFreePolicyInfo						*
CiGetPEInformation						
CiInitialize					*	
CiValidateFileObject		*	*	*		
CiVerifyHashInCatalog		*		*		
CiValidateImageHeader	*	*	*	*		
CiValidateImageData	*		*	*		
CiQueryInformation						*
CiSetFileCache		*	*	*		*
CiGetFileCache						*
CiHashMemory						*
KappxIsPackageFile						*
CiCompareSigningLevels						*
CiValidateFileAsImageType		*	*	*		

Function	Category					
CiRegisterSigningInformation						*
CiUnregisterSigningInformation						*
CiInitializePolicy					*	
SIPolicyQueryPolicyInformation						*
CiValidateDynamicCodePages[1]	*					
SIPolicyQuerySecurityPolicy						*
CiGetStrongImageReference						*
CiReleaseContext						*
CiHvciSetImageBaseAddress						*

1 Since this function verifies the integrity of pages that are dynamically generated ([Mi2018] Section 3.2.14), a detailed analysis is required to identify the origin and placement of the page hashes against which integrity is verified. Such analysis is out of the scope of this work.

Functionalities Exported by skci.dll

Function	Category					
	Image integrity verification				Initialization	Miscellaneous
	Image sections		Signature placement			
	Page	File	Embedded	Catalogs		
SkciCreateCodeCatalog						*
SkciCreateSecureImage						*
SkciFinalizeSecureImageHash						*
SkciFinishImageValidation	*	*	*	*		
SkciFreeImageContext						*
SkciInitialize					*	
SkciTransferVersionResource						*
SkciValidateDynamicCodePages	*[2]					
SkciValidateImageData	*		*	*		

2 Since this function verifies the integrity of pages that are dynamically generated ([Mi2018], Section 3.2.14), a detailed analysis is required to identify the origin and placement of the page hashes against which integrity is verified. Such analysis is out of the scope of this work.

ci.dll and skci.dll: Invocation Paths

ci.dll	Normal kernel	SSCN	Secure kernel	skci.dll
	VslpIumPhase4Initialize	0x1	SkInitSystem	SkciInitialize
	VslpIumPhase0Initialize	0xD0		
	VslMakeCodeCatalog	0x15	SkmmConvertSecureAllocationToCatalog	SkciCreateCodeCatalog
CiHvciCalculateHeaderHash CiHvciCalculateImageHash	VslCreateSecureImageSection	0x16	SkmmCreateSecureImageSection	SkciCreateSecureImage
CiHvciAddNonSectionDataToFileHash CiHvciCalculateImageHash CiHvciValidateImageData	VslValidateSecureImagePages	0xC1	SkmmValidateSecureImagePages	SkciValidateImageData
CiHvciTransferRelocationInformation	VslPrepareSecureImageRelocations	0x19	SkmmPrepareImageRelocations	
CiHvciValidateDynamicCodePages	VslValidateDynamicCodePages	0x1C	SkmmValidateDynamicCodePages	SkciValidateDynamicCodePages
CiHvciCalculateHeaderHash CiHvciCalculateImageHash	VslFinalizeSecureImageHash	0x17	SkmmFinalizeSecureImageHash	SkciFinalizeSecureImageHash
CiHvciVerifyFileHashInCatalogs → CipHvciVerifyHashInCatalogs CiHvciVerifyPageHashInCatalogs → CipHvciVerifyHashInCatalogs CiHvciVerifyFileHashSignedFile	VslFinishSecureImageValidation	0x18	SkmmFinishSecureImageValidation	SkciFinishImageValidation
	VslCreateSecureImageSection	0x16	SkmmCreateSecureImageSection	SkciFreeImageContext
		/	SkmiDeleteImage	
CiHvciSetFileVersionInformation	VslTransferSecureImageVersionResource	0x1D	SkmmTransferImageVersionResource	SkciTransferVersionResource

Event IDs

Microsoft-Windows-CodeIntegrity

Event ID	Event Message
3001	Code Integrity determined an unsigned kernel module %2 is loaded into the system. Check with the publisher to see if a signed version of the kernel module is available.
3002	Code Integrity is unable to verify the image integrity of the file %2 because the set of per-page image hashes could not be found on the system.
3003	Code Integrity is unable to verify the image integrity of the file %2 because the set of per-page image hashes could not be found on the system. The image is allowed to load because kernel mode debugger is attached.
3004	Windows is unable to verify the image integrity of the file %2 because file hash could not be found on the system. A recent hardware or software change might have installed a file that is signed incorrectly or damaged, or that might be malicious software from an unknown source.
3005	Code Integrity is unable to verify the image integrity of the file %2 because a file hash could not be found on the system. The image is allowed to load because kernel mode debugger is attached.
3006	Code Integrity found a set of per-page image hashes for the file %2 in a catalog %4.
3007	Code Integrity found a set of per-page image hashes for the file %2 in the image embedded certificate.
3008	Code Integrity found a file hash for the file %2 in a catalog %4
3009	Code Integrity found a file hash for the file %2 in the image embedded certificate.
3010	Code Integrity was unable to load the %2 catalog. Status %3.
3011	Code Integrity successfully loaded the %2 catalog.
3012	Code Integrity started loading the %2 catalog.
3013	Code Integrity started reloading catalogs.
3014	Code Integrity completed reloading catalogs. Status %1.
3015	Code Integrity completed reloading catalogs. Status %1.
3016	Code Integrity completed validating file hash. Status

	%1.
3017	Code Integrity started validating page hashes of %2 file.
3018	Code Integrity completed validating page hashes. Status %1.
3019	Code Integrity started loading catalog cache from %2 file
3020	Code Integrity completed loading catalog cache. Status %1.
3021	Code Integrity determined a revoked kernel module %2 is loaded into the system. Check with the publisher to see if a new signed version of the kernel module is available.
3022	Code Integrity determined a revoked kernel module %2 is loaded into the system. The image is allowed to load because kernel mode debugger is attached.
3023	Windows is unable to verify the integrity of the file %2 because the signing certificate has been revoked. Check with the publisher to see if a new signed version of the kernel module is available.
3024	Windows was unable to update the boot catalog cache file. Status %1.
3025	Code Integrity determined kernel module %2 is loaded into the system which does not have a valid embedded digital signature. Check with the publisher to see if an embedded signed version of the kernel module is available.
3026	Code Integrity was unable to load the %2 catalog because the signing certificate for this catalog has been revoked. This can result in images failing to load because a valid signature cannot be found. Check with the publisher to see if a new signed version of the catalog and images are available.
3027	Code Integrity started loading catalog %2 from the cache file.
3028	Code Integrity started saving catalog cache to %2 file.
3029	Code Integrity completed saving catalog cache. Status %1.
3030	Code Integrity saved catalog %2 to the cache file.
3032	Code Integrity determined a revoked image %2 is loaded into the system. Check with the publisher to see if a new signed version of the image is available.

3033	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements.
3034	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy. However, due to code integrity auditing policy, the image was allowed to load.
3035	Code Integrity determined a revoked image %2 is loaded into the system. The image is allowed to load because kernel mode debugger is attached.
3036	Windows is unable to verify the integrity of the file %2 because the signing certificate has been revoked. Check with the publisher to see if a new signed version of the kernel module is available.
3037	Code Integrity determined an unsigned image %2 is loaded into the system. Check with the publisher to see if a signed version of the image is available.
3038	Code Integrity started validating image header of %2 file.
3039	Code Integrity completed validating image header. Status %1.
3040	Code Integrity started retrieving the cached data of %2 file.
3041	Code Integrity completed retrieval of file cache. Status %1.
3042	Code Integrity started setting the cache of %2 file.
3043	Code Integrity completed setting the file cache. Status %1.
3050	Code Integrity completed retrieval of file cache. Status %1.
3051	Code Integrity completed retrieval of file cache. Status %1.
3052	Code Integrity completed retrieval of file cache. Status %1.
3054	Code Integrity started setting the cache of %2 file.
3055	Code Integrity completed setting the file cache. Status %1.
3057	Code Integrity completed retrieval of file cache. Status %1.
3058	Code Integrity completed retrieval of file cache. Status %1.

3059	Code Integrity found a set of per-page image hashes for the file %2 in a catalog %4.
3060	Code Integrity found a set of per-page image hashes for the file %2 in a catalog %4.
3061	Code Integrity found a file hash for the file %2 in a catalog %4.
3062	Code Integrity found a file hash for the file %2 in a catalog %4.
3063	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the security requirements for %5.
3064	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the security requirements for %5. However, due to system policy, the image was allowed to load.
3065	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the security requirements for %5. However, due to system policy, the image was allowed to load.
3066	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy. However, due to code integrity auditing policy, the image was allowed to load.
3067	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy. However, due to code integrity auditing policy, the image was allowed to load.
3068	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy.
3069	Code Integrity was unable to load the weak crypto policy value from registry. Status %1.
3070	Code Integrity was unable to load the weak crypto policy from registry store. Status %1.
3071	Code Integrity was unable to load the weak crypto policies. Status %1.
3072	Code Integrity determined that the kernel module %2 is not compatible with hypervisor enforcement due to it having non-page aligned sections.
3073	Code Integrity determined that the kernel module %2 is not compatible with strict mode hypervisor enforcement due to it having an executable section that is also writable.

3074	Code Integrity was unable to verify a page for a module verified using hypervisor enforcement. Status %1.
3075	Code Integrity determined that process (%4) spent %7 and %8 microseconds for Code Integrity check and policy check to load %2 with validated %6 signing level. For all components without EA cache, Code Integrity spent about %9ë more time when policy enforced.
3076	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy. However, due to code integrity auditing policy, the image was allowed to load.
3077	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy.
3078	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy. However, due to code integrity auditing policy, the image was allowed to load.
3079	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy.
3080	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy. However, due to code integrity auditing policy, the image was allowed to load.
3081	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the %5 signing level requirements or violated code integrity policy.
3082	Code Integrity determined kernel module %2 that did not meet the WHQL requirements is loaded into the system. However, due to code integrity auditing policy, the image was allowed to load.
3083	Code Integrity determined kernel module %2 that did not meet the WHQL requirements is loaded into the system. Check with the publisher to see if a WHQL compliant kernel module is available.
3084	Code Integrity will enable WHQL driver enforcement for this boot session. Settings %1. Exemption %2.

3085	Code Integrity will disable WHQL driver enforcement for this boot session. Settings %1.
3086	Code Integrity determined that a process (%4) attempted to load %2 that did not meet the signing requirements for Isolated User Mode.

Microsoft-Windows-Applocker

Event ID	Event Message
8029	%2 was prevented from running due to Config code integrity (CI) policy.
8036	%2 was prevented from running due to Config CI policy.

Microsoft-Windows-DeviceGuard

Event ID	Event Message
7000	Device Guard successfully processed the Group Policy: Virtualization Based Security = %1, Secure Boot = %2, DMA Protection = %3, Virtualization Based Code Integrity = %4, Credential Guard = %5, Reboot required = %6, Status = %7.
7001	Device Guard failed to process the Group Policy to enable Virtualization Based Security (Status = %1): %2
7002	Device Guard failed to process the Group Policy to disable Virtualization Based Security (Status = %1): %2
7010	Device Guard successfully processed the Group Policy: Configurable Code Integrity Policy = %1, Policy file path = %2, Reboot required = %3, Status = %4.
7011	Device Guard failed to process the Group Policy to enable Configurable Code Integrity Policy (Status = %1): %2
7012	Device Guard failed to process the Group Policy to disable Configurable Code Integrity Policy (Status = %1): %2
7013	Device Guard is not available in this edition of Windows

Reference Documentation

ERNW WP6	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 6
ms_dg	https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/ [17/7/2018]
ERNW WP5	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 5
Yosif 2017	Yosifovic, Pavel; Ionescu, Alex; Russinovich, Mark E.; Solomon, David A. : Windows Internals, Part 1 and Part 2
rfc_pkcs	https://tools.ietf.org/html/rfc2315 [13/9/2018]
ms_memcmp	https://msdn.microsoft.com/de-de/library/zyaebf12.aspx [20/7/2018]
ms_str	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-rtlequalunicodestring [20/7/2018]
ms_appl	https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-overview [11/9/2018]
ERNW WP2	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 2
ms_inteli	http://cloud-platform-assets.azurewebsites.net/intelligent-security-graph/ [17/7/2018]
ms_newci	https://docs.microsoft.com/en-us/powershell/module/configci/new-cipolicy?view=win10-ps [17/7/2018]
ms_rules	https://docs.microsoft.com/en-us/windows/device-security/device-guard/deploy-code-integrity-policies-policy-rules-and-file-rules#code-integrity-file-rule-levels [17/7/2018]
ms_whlk	https://docs.microsoft.com/en-us/windows-hardware/test/hlk/windows-hardware-lab-kit [17/7/2018]
ms_whqlsig	https://docs.microsoft.com/en-us/windows-hardware/drivers/install/whql-release-signature [17/7/2018]
ms_insider	https://insider.windows.com/en-us/ [17/7/2018]
ms_sscm	http://download.microsoft.com/download/5/D/B/5DBEBA38-8D5D-4119-B2E8-B8369B74BF43/system_center_configuration_manager_and_microsoft_intune_datasheet.pdf [17/7/2018]
ms_auth	http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx [18/7/2018]
ms_embcat	https://docs.microsoft.com/en-us/windows-hardware/drivers/install/authenticode [18/7/2018]
Mi2018	Microsoft: Microsoft Windows FIPS 140 Validation
Russinovich 2012	Russinovich, Mark E.; Solomon, David A.; Ionescu, Alex: Windows Internals, Part 2
ERNW WP4	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 4
ms_bsrc	https://msdn.microsoft.com/en-us/library/w0k41tbs.aspx [20/7/2018]
rfc_tbs	https://tools.ietf.org/html/rfc5280#section-4.1.1.1 [20/7/2018]
ms_sccm	https://docs.microsoft.com/en-us/sccm/protect/deploy-use/use-device-guard-with-configuration-manager [12/9/2018]
ms_gpo	https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/deploy-windows-defender-application-control-policies-using-group-policy [12/9/2018]
ms_intune	https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/deploy-windows-defender-application-control-policies-using-intune [12/9/2018]

ms_pscci	https://docs.microsoft.com/en-us/powershell/module/configci/?view=win10-ps [12/9/2018]
ms_mrg	https://docs.microsoft.com/en-us/powershell/module/configci/merge-cipolicy?view=win10-ps [12/9/2018]
ms_conv	https://docs.microsoft.com/en-us/powershell/module/configci/convertfrom-cipolicy?view=win10-ps [12/9/2018]
ms_sro	https://docs.microsoft.com/en-us/powershell/module/configci/set-ruleoption?view=win10-ps [12/9/2018]
ms_addsr	https://docs.microsoft.com/en-us/powershell/module/configci/add-signerrule?view=win10-ps [12/9/2018]
ms_arb	https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/microsoft-recommended-block-rules [11/9/2018]
ms_al	https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load?redirectedfrom=MSDN&view=netframework-4.7.2#System_Reflection_Assembly_Load_System_String_ [11/9/2018]
ERNW WP8	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10: Work Package 8
ERNW WP9	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 9
ms_pmix	https://blogs.technet.microsoft.com/datacentersecurity/2018/03/10/default-code-integrity-policy-for-windows-server/ [13/9/2018]
ms_log	https://docs.microsoft.com/en-us/sccm/protect/deploy-use/use-device-guard-with-configuration-manager [17/8/2018]

Keywords and Abbreviations