Federal Office
for Information Security

Device Setup Manager

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

Code Blocks

1 Introduction

1.1 Zusammenfassung

Dieses Dokument stellt das Ergebnis von Arbeitspaket 3d des Projekts „SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ dar. Das Projekt wird durch die Firma ERNW Enno Rey Netzwerke GmbH im Auftrag des Bundesamts für Sicherheit in der Informationstechnik (BSI) durchgeführt.

Ziel dieses Arbeitspakets ist eine Analyse des Windows Dienstes `Device Setup Manager` (`DsmSvc`) des Microsoft Windows 10 Betriebssystems. Wie durch das BSI vorgegeben, wird Windows 10 Enterprise, long-term servicing channel (LTSC) 2019, Deutsch betrachtet.

Die wesentlichen Inhalte dieser Arbeit sind:

- Überblick über die Plug-and-Play- (PnP) Infrastruktur: Zunächst werden die wesentlichen Konzepte im Zusammenhang mit der PnP-Infrastruktur definiert. Dazu gehört die Vorstellung der einzelnen Komponenten und ihrer Beziehungen.
- Analyse des `DsmSvc`: Die Arbeit liefert eine detailliertere Darstellung der inneren Arbeitsweise von `DsmSvc`. Der Fokus liegt auf dem Starten und Initialisieren des Dienstes, sowie dem Verfahren, das `DsmSvc` intern verwendet, um eine Geräteinstanz zu unterstützen. In der Praxis bedeutet dies z. B. das Suchen, Herunterladen und Bereitstellen eines fehlenden Gerätetreibers.

Die Arbeit sowie die entsprechenden Ergebnisse stellen sich wie folgt dar:

Begriffe und Konzepte (Kapitel 1.3) Dieses Kapitel führt allgemeine Konzepte und Begriffe im Zusammenhang mit `DsmSvc` und der PnP-Infrastruktur ein, die für das Verständnis der Arbeit relevant sind. Hierzu gehören:

- **Überblick über PnP-Infrastruktur** (Kapitel 1.3.1) Modernen Computersystemen steht eine Vielzahl von Geräten zur Verfügung, die im Alltag eine entscheidende Rolle spielen. Diese Geräte können von einfachen Eingabe-/Ausgabe-Peripheriegeräten wie Tastaturen und Mäusen bis hin zu komplexeren Geräten wie Netzwerkkarten und Festplattenlaufwerken reichen. Unabhängig von ihrem Typ, ihrer Funktionalität und der Art ihrer Anbindung an das System müssen alle diese Geräte jedoch in der Lage sein, effizient mit dem System zu kommunizieren.

  Hier kommt das Input/Output (I/O)-System ins Spiel, das als Schnittstelle zwischen dem Gerät und dem Kernel des Windows-Betriebssystems dient. Bestehend aus mehreren Komponenten stellt das I/O-System sicher, dass Daten zwischen einem Gerät und dem Betriebssystem übertragen werden können.

  Das I/O-System besteht aus den folgenden Komponenten, die sich auf Userland und Kernelland verteilen:

  - Hardware Abstraction Layer (HAL) (Kapitel 1.3.1.1): Der HAL ist ein Kernel-Modul, der eine direkte Schnittstelle zur Hardware darstellt und in der Datei `hal.dll` implementiert ist. Diese Schnittstelle abstrahiert architekturspezifische Details der Hardware von dem Windows-Betriebssystem.
  - Gerätetreiber (Kapitel 1.3.1.2): Gerätetreiber sind Kernel-Module (typischerweise mit der Dateiendung `.sys`), die geladen werden können und eine Schnittstelle zwischen Kernel und Hardware darstellen. Tatsächlich kommunizieren jedoch auch Treiber nicht direkt mit der Hardware, sondern durch den HAL.
  - PnP-Manager (Kapitel 1.3.1.3): Der PnP-Manager ist eine wesentliche Komponente des Windows-Betriebssystems, der für die Erkennung und Verwaltung von Geräten verantwortlich ist. Zusammenfassend ist der PnP-Manager für drei Hauptaufgaben verantwortlich:

        - Geräteerkennung während des Systemstarts.
        - Verarbeitung des Hinzufügens oder Entfernens von Geräten, während das System läuft.
        - Installation und Einrichtung neuer Geräte mit einem geeigneten Treiberpaket.
    - Userland PnP-Manager (Kapitel 1.3.1.4): Der Userland PnP-Manager muss als Konzept und nicht als unabhängige Komponente betrachtet werden. Er ist eine privilegierte Userland-Identität, die mit ihrem Kernelland-Gegenstück (d.h. Kernelland PnP-Manager) interagiert. Der Userland PnP-Manager bietet übergeordnete Schnittstellen zu anderen Userland-Komponenten und ermöglicht zusätzliche PnP-Funktionalität. Auf diese Weise bietet er eine definierte Möglichkeit, für Userland-Komponenten (z. B. Windows Update), eine Schnittstelle mit der PnP-Infrastruktur bereitzustellen.
    - Dienste (Kapitel 1.3.1.5): Es gibt zahlreiche Dienste, die PnP-Funktionalität entweder direkt implementieren oder im Kontext von PnP genutzt werden, ohne ausschließlich dafür konzipiert zu sein. Ein Beispiel für einen Dienst, der ausschließlich im Zusammenhang mit PnP entwickelt wurde, ist `DsmSvc`. Er ist für die Installation von Gerätetreiberpaketen und zugehöriger Software verantwortlich. Dabei koordiniert und verteilt er die eigentlichen Aufgaben. Zum Beispiel an den Windows Update-Dienst. Dieser Dienst ist – anders als `DsmSvc` - nicht ausschließlich auf die Unterstützung von PnP ausgelegt, sondern kann auch in anderen Zusammenhängen eingesetzt werden.
    - Setupkomponenten (Kapitel 1.3.1.6): Beim Einrichten von Gerätetreibern stehen mehrere Setupkomponenten zur Verfügung, um den Vorgang zu vereinfachen. Sie bieten eine Reihe von Funktionen und Tools, die die Installation und Verwaltung von Gerätetreibern erleichtern. Dazu gehören beispielsweise Aufgaben wie: Lesen und Verarbeiten von Dateien und anderen gerätebezogenen Objekten im System.
    - Anwendungen (Kapitel 1.3.1.7): Windows stellt eine Vielzahl von Anwendungen bereit, die dabei helfen können, Geräte in Windows anzuzeigen und zu verwalten. Eine solche Anwendung ist das Dienstprogramm Device Manager[1]. Der Device Manager verwendet Setupkomponenten, um mit den Geräteinstallations- und Konfigurationsfunktionalitäten des Betriebssystems zu interagieren. Dadurch können Aufgaben wie das Anzeigen von Geräteeigenschaften, das Aktualisieren von Treibern, das Deaktivieren oder Deinstallieren von Geräten sowie die Fehlerbehebung bei gerätebezogenen Problemen ausgeführt werden.
    - Treiberpaket (engl. driver package) und Treiberspeicher (engl. Driver Store) (Kapitel 1.3.1.8): Treiberpaket und Treiberspeicher sind zwei Konzepte, die sich auf die Gerätetreiberverwaltung beziehen. Ein Treiberpaket ist eine Sammlung von Softwarekomponenten, die die Integration eines Geräts in das I/O System ermöglicht.

      Der Treiberspeicher dient als vertrauenswürdige zentrale Quelle für die von einem Treiberpaket bereitgestellten Softwarekomponenten. Während des Treiberinstallationsprozesses können nur Ressourcen verwendet werden, die sich im Treiberspeicher befinden. Dabei muss zwischen zwei unterschiedlichen Prozessen unterschieden werden: Staging und Installation. Staging definiert den Prozess des Kopierens eines Treiberpakets in den Treiberspeicher. Die Installation definiert die vollständige Instanziierung aller relevanten Ressourcen für das Treiberpaket, so dass der Treiber zusammen mit dem ihm zugeordneten Gerät vom Betriebssystem verwendet werden kann.
- **Geräteinstanz** (Kapitel 1.3.2): Im Kontext von PnP bezieht sich der Begriff „Geräteinstanz“ auf ein instanziiertes Gerät im Windows-Betriebssystem. Technisch gesehen verwaltet der im Kernel

[1] https://learn.microsoft.com/en-us/troubleshoot/windows-server/deployment/use-device-manager-configure-devices [Abgerufen am 3/3/2023]

implementierte PnP-Manager eine Datenstruktur, die als Gerätebaum (engl. Device Tree) bekannt ist. Beim Systemstart wird diese Datenstruktur initialisiert und jedes Mal, wenn ein neues Gerät angeschlossen oder entfernt wird, wird diese Datenstruktur aktualisiert.

Der Gerätebaum besteht aus Geräteknoten (engl. Device Nodes), die vorhandene PnP-fähige Geräte referenzieren. Ein Geräteknoten wiederum besteht aus allen instanziierten Informationen (d. h. Datensätzen), die zusammen ein Gerät im Windows-Betriebssystem darstellen. Dazu gehören die Geräteobjekte, die das Gerät darstellen, sowie andere relevante Informationen im Zusammenhang mit PnP, wie z. B. Statusflags für Geräteknoten[2]. Letztlich bezieht sich der Begriff „Geräteinstanz" auf einen im Gerätebaum angelegten Geräteknoten.

- **Geräteidentifikationszeichenfolgen** (engl. Device identification strings) (Kapitel 1.3.3): Eine Geräteidentifikationszeichenfolge ist eine eindeutige Kennung, die entweder vom Hersteller während des Herstellungsprozesses oder vom Betriebssystem selbst, typischerweise während der Installation oder des Startvorgangs des Geräts, zugewiesen werden kann. Sie besteht normalerweise aus einer Kombination aus Buchstaben, Zahlen und Symbolen, die für das Gerät und seinen Treiber spezifisch sind. Einem Gerät können mehrere Identifikationszeichenfolgen zugeordnet sein. Allerdings ist jede dieser Identifikationszeichenfolgen, einschließlich der Geräte-ID, der Hardware-Ids, der kompatiblen Ids, der Instanz-ID und der Container-ID, eindeutig und dient der Identifizierung eines Geräts in verschiedenen Kontexten.
- **Geräteeigenschaftsmodell** (Kapitel 1.3.4): Der PnP-Manager stellt das Geräteeigenschaftsmodell[3] (engl. Unified Device Property Model) zur Verfügung. Dieses Modell bietet eine standardisierte Vorgehensweise zum Speichern und Abfragen von Informationen über Geräteinstanzen. In der Praxis definiert es die Darstellung von Geräteeigenschaften, die die Geräteinstanzen beschreiben, und stellt eine einheitliche Schnittstelle für den Zugriff darauf bereit.
- **Verfahren zur Geräteidentifizierung** (Kapitel 1.3.5): Dieses Kapitel beschreibt eine Übersicht über die Identifizierung eines Geräts im Kontext von PnP, wenn ein USB-Stick in Windows eingesteckt wird. Dieses Kapitel basiert auf der Originaldokumentation von Microsoft[4] und beschreibt jeden Schritt des Prozesses detailliert, um ein besseres Verständnis dafür zu vermitteln, wie die Geräteidentifizierung im Zusammenhang mit PnP funktioniert.

Analyse von DsmSvc (Kapitel 2): `DsmSvc` ist ein Dienst, der in die Plug and Play Infrastruktur von Windows eingebunden ist. Der Dienst ist ein Taskscheduler, der bei Bedarf durch den Plug and Play Manager gestartet wird. Dies geschieht, wenn ein Gerät an das System angeschlossen wird und einen Gerätetreiber benötigt. Die Aufgabe von `DsmSvc` ist in diesem Zusammenhang, die Aufgaben (Tasks), die dafür nötig sind, vorzubereiten und durchzuführen bzw. durchführen zu lassen. Der Dienst ist auf andere Komponenten angewiesen, um einerseits Daten abzufragen und andererseits die eigentliche Arbeit auszuführen. Daher spielt die Kommunikation eine entscheidende Rolle. In diesem Zusammenhang ist der Dienst dafür verantwortlich, die zugrunde liegende Aufgabe zu planen, also mit verschiedenen Komponenten (z.B. Windows Update) zu kommunizieren, die letztendlich die eigentliche Arbeit (z.B. Herunterladen von Treibern) ausführen.

In den darauffolgenden Abschnitten werden Teile der inneren Arbeitsweise des `DsmSvc` vorgestellt. Dabei liegt der Fokus auf dem Starten und der Initialisierung des Diensts, sowie dem Verfahren, das `DsmSvc` intern verwendet, z.B. zum Suchen, Herunterladen und Bereitstellen von Gerätetreibern.

[2] https://learn.microsoft.com/de-de/windows-hardware/drivers/debugger/device-node-status-flags [Abgerufen am 3/3/2023]

[3] https://learn.microsoft.com/de-de/windows-hardware/drivers/install/unified-device-property-model--windows-vista-and-later- [Abgerufen am 3/3/2023]

[4] https://learn.microsoft.com/de-de/windows-hardware/drivers/kernel/adding-a-pnp-device-to-a-running-system [Abgerufen am 3/3/2023]

- **Start des Dienstes** (Kapitel 2.1) `DsmSvc` ist ein Windows Dienst, der nur durch einen bestimmten Auslöser (engl. trigger) gestartet wird.[5] Dieser Auslöser kann für jeden Dienst in der Registry konfiguriert werden. In diesem Fall wird der Auslöser vom PnP-Manager (im Kernelland) getätigt, wenn ein Gerät (z.B. ein USB-Stick) in das System gesteckt wird und dieses Gerät einen Gerätetreiber benötigt. Wird der Dienst gestartet, hat dieser keine Information, welches Gerät Unterstützung benötigt. Der Dienst weiß nur, dass ein ihm noch unbekanntes Gerät Unterstützung benötigt.
- **Initialisierung des Dienstes** (Kapitel 2.2) Wenn der Dienst startet, wird dieser zunächst initialisiert. Hierbei wird die Konfiguration des Dienstes aus der Registry gelesen. Weiterhin werden wichtige Komponenten bereitgestellt, die der Dienst benötigt. Dabei handelt es sich einerseits um den Thread Pool, der später verwendet wird, um die anfallenden Aufgaben abzuarbeiten. Andererseits wird eine Callback-Funktion registriert, die verwendet wird, um zu bestimmen, welche Geräte Unterstützung benötigen.
- **Initialisierung der Schedulerqueue** (Kapitel 2.3) Der PnP-Manager nutzt die Callback-Funktion, um den Dienst zu informieren, welche Geräte Unterstützung benötigen. Basierend auf diesen Informationen werden dann die Jobs und Tasks erstellt, um die Unterstützung sicherzustellen. Jedes Gerät erhält eine eigene Job Queue, in jedem dieser Jobs sind die Tasks für diesen Job enthalten. Beispielsweise, beinhaltet der `START_DEVICE_INSTALL` Job den `InitializeInstallTask`. Ein Job subsumiert also einzelne Tasks, in diesen Tasks findet dann die eigentliche Arbeit statt.
- **Download- und Staging-Prozedur** (Kapitel 2.4) Generell können die benötigten Treiber entweder durch eine lokale oder eine online Prozedur bereitgestellt werden. Im nachfolgenden wird nur die online Prozedur betrachtet. `DsmSvc` nutzt hierbei Windows Update, um nach Gerätetreibern zu suchen, diese herunterzuladen und zu stagen. `DsmSvc` selbst ist an diesem Prozess nicht primär beteiligt, sondern delegiert die Arbeit, die dann im Rahmen von Windows Update ausgeführt wird.

Konfiguration und Logging (Kapitel 3) `DsmSvc` kann via Gruppenrichtlinie unter den folgenden Pfaden konfiguriert werden: `Computerkonfiguration` → `Administrative Vorlagen` → `System` → `Geräteinstallation` und `Benutzerkonfiguration` → `Administrative Vorlagen` → `System` → `Geräteinstallation`.

Kapitel 3.2 stellt Event Tracing for Windows (ETW) Provider für das Logging von `DsmSvc`-bezogenen Ereignissen vor. Table 1 zeigt die Namen und GUIDs (globally unique identifiers) der ETW-Provider, die in `DsmSvc` implementiert sind.

ETW Provider	GUID
`Microsoft-Windows-DeviceSetupManager`	`FCBB06BB-6A2A-46E3-ABAA-246CB4E508B2`
`Microsoft.Windows.DeviceSetupManager`	`AB11A476-79F6-5026-7D54-2E9B9E539A2D`

Table 1: ETW Provider in DsmSvc

Zusammenfassung:

Die Implementierung von PnP basiert auf einer komplexen Infrastruktur, die sich über viele Bereiche des Betriebssystems erstreckt, einschließlich Kernelland und Userland. Darüber hinaus kommen verschiedene Kommunikationsmethoden zum Einsatz. Erwähnenswert ist, dass die breite Verteilung dieser Technologie sowohl auf historische als auch strukturelle Faktoren zurückzuführen ist. Aus sicherheitstechnischer Sicht ist jedoch ein positiver Aspekt, dass das Prinzip der Funktionstrennung durch die Verteilung der Komponenten, eingehalten wird.

Im Kontext von PnP fungiert der `DsmSvc` Dienst in erster Linie als Verteilungspunkt und fungiert als Taskscheduler, der die Installation und Konfiguration von Hardwaregeräten und Treibern automatisiert, sodass kein manueller Eingriff erforderlich ist. Hierbei ist der Dienst jedoch auf andere Komponenten

[5] https://learn.microsoft.com/en-us/windows/win32/services/service-trigger-events [Abgerufen am 3/3/2023]

angewiesen, um Aufgaben wie das Abrufen von Daten abzuarbeiten, die für die Durchführung der automatisierten Prozesse von entscheidender Bedeutung sind.

Wie bereits beschrieben, funktioniert die PnP-Technologie, unterstützt durch ihre `DsmSvc` Komponente, automatisch ohne jeglichen Benutzereingriff, was sowohl Vor- als auch Nachteile haben kann. Die Technologie ermöglicht die automatische Installation einiger Treiber aus dem Treiberspeicher, ohne dass Administratorrechte erforderlich sind. Wenn jedoch unbefugter Zugriff auf den Treiberspeicher erlangt wird, kann ein bösartiger Treiber installiert werden, der die Sicherheit des gesamten Systems gefährdet. Da der Prozess automatisiert und transparent ist, ist die Nachverfolgung böswilliger Aktivitäten in diesem Zusammenhang eine Herausforderung. Auch wenn ein Treiber ohne Benutzerinteraktion geladen werden kann, muss der Treiber dennoch über eine gültige Signatur verfügen.

Zusammenfassend lässt sich sagen, dass die Verwendung dieses automatischen Prozesses einen Kompromiss zwischen Benutzerfreundlichkeit und Sicherheit darstellt. Um eine bessere Kontrolle über den Prozess zu haben, kann man entweder diesen automatischen Prozess deaktivieren oder eine Härtung des Treiberspeichers in Betracht ziehen. Hier könnte das Härten des Treiberspeichers konzeptionell dem Prozess des Härtens des Stammzertifikatspeichers ähneln.

1.2 Executive Summary

This document implements the work plan outlined in Work Package AP3d DsmSvc of the project "SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10" (orig., ger.). The project is carried out by the company ERNW Enno Rey Netzwerke GmbH on behalf of the German Federal Office for Information Security (orig., ger., Bundesamt für Sicherheit in der Informationstechnik - BSI).

The objective of this work package is the analysis of the `Device Setup Manager` Service (`DsmSvc`) of Windows 10. As required by the German Federal Office for Information Security, the release of the Windows 10 system in focus is Windows 10 Enterprise, long- term servicing channel (LTSC) 2019, German language.

The core contributions of this work are:

- An overview of the Plug and Play Infrastructure: First the essential concepts related to the Plug and Play Infrastructure are defined. This includes discussions on the relationship between the components.
- An analysis of `DsmSvc` service. It introduces a more detailed view of the "inner workings" of the `DsmSvc`. The focus is on starting and initial initialization of the service as well as the procedure that `DsmSvc` uses internally to support a device instance. In practice this means e.g., searching, downloading, and staging of a missing device driver.

This work is structured as follows:

- Section 1.3 introduces general concepts and terms related to `DsmSvc` and the Plug and Play infrastructure which are relevant to understand what is presented.
- Section 2 provides technical information on functionalities, whose analysis is in the scope of this work package.
    - Section 2.1 discusses trigger started Windows services. It is presented where trigger information is stored, and the relevant trigger used for starting `DsmSvc`.
    - Section 2.2 describes the initialization of `DsmSvc`.
    - Section 2.3 analyzes the task system used in `DsmSvc`. It discusses the different schedulers and their relationship to jobs and tasks.
    - Section 2.4 provides and overview of how a particular task is executed in the context of `DsmSvc`. In this case the online retrieval and staging procedure of driver packages is presented.
- Section 3 provides an overview of the configuration and logging capabilities of Windows 10 for managing `DsmSvc` and logging relevant events.

Note: Throughout the technical discussions, the analysis presented in this work was performed by applying static and dynamic code analysis methods using the `WinDbg` debugger and the IDA disassembler. The technical discussions in this work include depictions of function call stacks and pseudo-code. These call stacks, for the sake of brevity, present only functions that are relevant to the discussions. The depicted pseudo-code is a high abstraction of real code, and it does not consistently follow the syntax of a particular programming language. It loosely follows a C and C++-like programming language syntax.

Summary

The Plug and Play (PnP) technology is a comprehensive and complex infrastructure that extends across many aspects of the operating system, including kernel-land and user-land. PnP also leverages various communication methods, such as COM and RPC. It is worth mentioning that the widespread implementation of this technology can be attributed to both historical and structural factors. However, a

positive aspect of the broad distribution of this technology is – from a security point of view - that it incorporates a principle of separation of duties.

In the context of the PnP technology, `DsmSvc` primarily functions as a distribution point, acting as a task scheduler that automates the installation and configuration of hardware devices and drivers, thus removing the need for manual intervention. However, `DsmSvc` is reliant on other components to perform tasks like data fetching, which are crucial for completing the automated processes.

The PnP technology, with its component `DsmSvc`, works automatically without any user intervention, which can have both advantages and disadvantages. The technology enables some drivers to be installed automatically from the Driver Store without the need for direct administrative privileges. However, if unauthorized access is gained to the driver store, a malicious driver could be installed that may compromise the system's security. As the process is automated and transparent, tracking any malicious activity would be challenging. Even though a driver can be loaded without user interaction, the driver must still have a valid signature.

In summary, utilizing this automatic process involves a tradeoff between usability and security. To have greater control over the process, one may consider either disabling this aspect of the technology or implementing Driver Store hardening, conceptually similar to the process used to harden the root certificate store.

1.3 Concepts and Terms

1.3.1 Plug and Play Infrastructure

In a modern computing environment, a wide variety of devices are available that play a vital role in the daily computing experience. These devices can range from simple input/output peripherals such as keyboards and mice, to more complex ones such as network cards and disk drives. However, regardless of their type, functionality, and the way they are connected to the system, all of these devices must be able to communicate effectively with the software running on the system in order to function correctly.

This is where the Input/Output (I/O) System comes in, serving as the interface between the device and the Windows operating system's kernel. Composed of several components, the I/O System ensures that data is transferred seamlessly and efficiently between the device and the operating system, making it a crucial aspect of modern computing environments. One of these components is Plug and Play (PnP), a technology that allows devices to be attached to the system and automatically configured without the need for manual intervention. With PnP, devices can be easily added or removed from a system without causing any disruption, making it a crucial component of the I/O System.

Figure 1 provides a conceptual overview of the various PnP components (also referred to as PnP infrastructure) involved in Windows when interacting with a device.[6] The components are described in the subsequent sections, starting from the bottom of the figure, and moving towards the top. This description outlines the roles and functions of each component and how they work together to ensure that devices are properly recognized, configured, and managed by the operating system.

[6] https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/pnp-components [Retrieved: 3/3/2023]

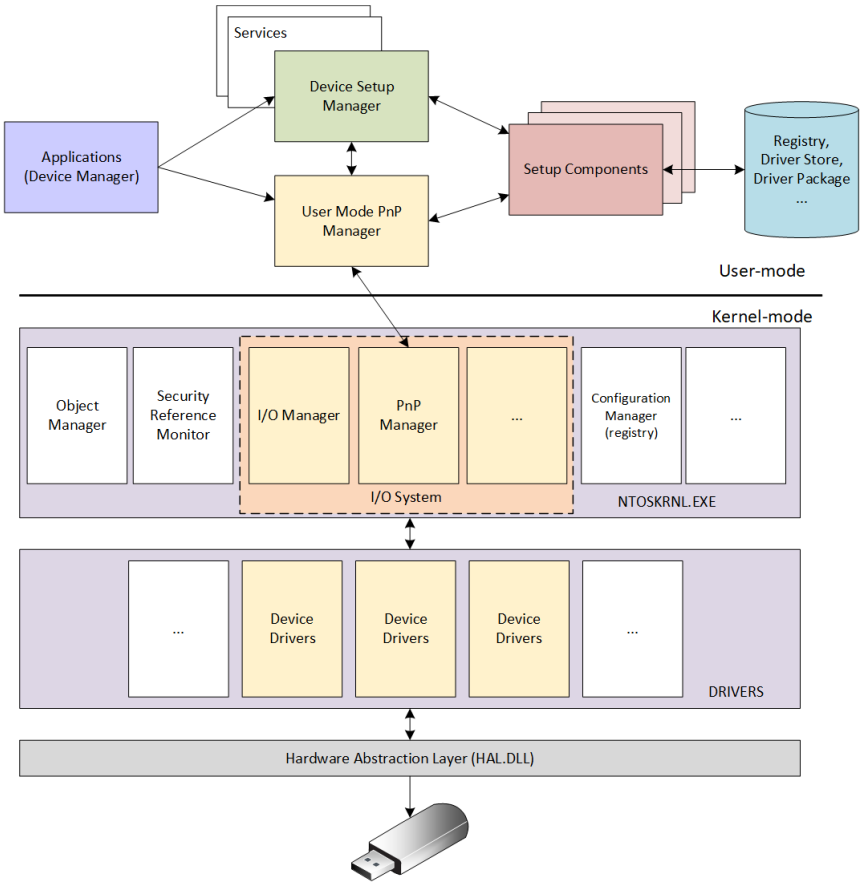

Figure 1: Conceptual view of the kernel of Windows, specifically on the I/O System dealing with a device.

1.3.1.1 Hardware Abstraction Layer (HAL) Library

The Hardware Abstraction Layer (HAL)[7] is an essential component of the Windows operating system, serving as the closest piece of software to the hardware. Its primary function is to facilitate interaction with hardware while abstracting low-level hardware details from device drivers and the operating system. The HAL provides a generic set of routines that can interface with different platforms where Windows is supported, including x86, x64 and ARM. As such, the HAL can be seen as the lowest-level device driver for all devices that are not directly managed by other drivers. By providing this level of abstraction, the HAL allows drivers and the operating system to interact with the hardware in a standardized and consistent manner, regardless of the specific hardware platform. This approach simplifies the development and maintenance of device drivers and operating system, making it easier for hardware manufacturers to create drivers that are compatible with Windows.

1.3.1.2 Device Drivers

Device drivers are loadable kernel modules (typically with the extension `.sys`) that serve as an interface between the kernel's executive functionalities managing input/output (I/O) and hardware. These drivers are divided into several categories based on their specific purpose, including interfacing with the Plug and Play (PnP) and I/O Manager architecture, as well as directly handling the device.

The first category is bus drivers, who are responsible for managing communication with any device sharing the same bus communication technology. They provide a standard interface for the devices to communicate with the operating system and manage the data flow between devices and the computer. The second

[7] https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-hal-library [Retrieved: 3/3/2023]

category is function drivers, which are dedicated to a particular device and are responsible for handling it from a functional perspective. For example, a USB stick's function driver would be responsible for managing its storage capabilities, while a keyboard's function driver would interpret the keystrokes. The third category is filter drivers, which serve as an intermediary between function and bus drivers to provide additional value regarding specific interactions. They can be used to modify data transmitted between a device and the operating system, providing enhanced security or performance optimization.

1.3.1.3 PnP Manager

The Plug and Play (PnP) Manager is an essential component of the Windows operating system responsible for detecting and managing devices on a system. It is implemented within the kernel executive, which allows it to perform tasks such as device detection and enumeration during system boot-up. Additionally, the PnP Manager handles the insertion or removal of devices while the system is running, allowing for seamless device management.

From a conceptual standpoint, the PnP Manager initiates the procedure to install and set up a driver for a new device inserted for the first time. The procedure of installing and setting up a driver for a new device is split between user-land and kernel-land. However, if the setup procedure is initiated in kernel-land, the installation phase is subsequently carried out in user-land. For example, the Device Setup Manager service operating in user-land registers itself to be notified about potential devices needing support to perform the installation phase with the assistance of various setup components.

In summary, the PnP Manager is responsible for three primary Plug and Play (PnP) related tasks:

- Device detection and enumeration while the system is booting.
- Processing addition or removal of devices while the system is running.
- Managing the process of installing and setting up new devices with a corresponding driver package.

1.3.1.4 User-land PnP Manager

The user-land PnP Manager is more considered as a concept rather than an independent component, allowing its functionality to be shared among different services or libraries depending on the individual situational context. However, this work defines it as a privileged user-land component that interfaces with its kernel-land counterpart. This offers higher-level interfaces to other user-land components enabling additional PnP functionality. That way, it provides a convenient way for user-land components (i.e., Windows Update) to interface with the PnP.

In practice, the user-land PnP Manager communicates with the kernel-land PnP Manager, which manages devices in the system by detecting, configuring, and monitoring them. The user-land PnP Manager receives notifications about device changes from the kernel-land PnP Manager, such as adding or removing a device, and takes appropriate action. It also provides a user-friendly interface for configuring and managing devices, allowing user-land entities to operate device settings and resolve compatibility and functionality issues.

For example, the user-land PnP Manager coordinates the installation of missing device drivers and associated software, working closely with the kernel-land PnP Manager to ensure proper configuration and management of the device. Acting as an intermediary between the user-land and kernel-land components, the user-land PnP Manager helps to ensure the smooth operation of the device and its software components.

1.3.1.5 Services

Numerous services either directly implement PnP functionality or are used in the context of PnP without being designed exclusively for it. One example of a service designed exclusively to support PnP is `DsmSvc`, which is responsible for coordinating tasks necessary to install software components and ensure proper

device function. Specifically, `DsmSvc` manages the installation process of device driver packages and associated software, collaborating with the kernel-mode PnP Manager to ensure proper device configuration and management.

Another example would be the Windows Update service, which is not exclusively designed to support PnP but is used in the context of it. This happens when the PnP manager detects a device for which no software is available and which should be made available via Windows Update.

1.3.1.6 Setup components

When setting up drivers for devices, several setup components are available to simplify the process. It must be emphasized that the multitude of setup components is partly due to the need for backward compatibility and the variety of methods available for maintaining a device within the system.

In practice, setup components provide a set of functions and tools that facilitate the installation and management of device drivers, as well as other tasks like reading and processing files, registry keys, and other objects in the system that are related to devices. The Windows operating system primarily utilizes these components during setup and installation. However, they can also be accessed by third-party applications that require access to the system's device management functionality.

1.3.1.7 Applications

There is a wide variety of applications available for Windows that can help to view and manage devices on the system. One such application is the Device Manager utility[8]. The Device Manager uses setup components to interact with the device installation and configuration functionality of the operating system. This enables it to perform tasks such as displaying device properties, updating drivers, disabling or uninstalling devices, and troubleshooting device-related issues. With its user-friendly graphical interface, the Device Manager makes it easy for users to manage their devices without directly interacting with the underlying I/O system.

1.3.1.8 Driver Package and Driver Store

Driver Package and Driver Store are two concepts that are related to device driver management in Windows. A driver package is a collection of software components that enable a device to integrate in the I/O system. Typically, a driver package includes one or more drivers that support various device functions, as well as configuration files, utilities, and other resources that the device requires. These packages are crucial for ensuring that devices work correctly and are critical to the overall device installation process.

As a brief description of the content of a driver package, one typically contains the following components:[9]

- **INF file** is mandatory for every driver package. This file describes how a driver package must be installed on a device, including driver files, registry entries, device IDs, catalog files, and version information.[10]
- **Catalog file** is referenced in the “INF Version section”[11] of the INF file. A catalog file contains a cryptographic hash for each file in the driver package. These hashes are used by Windows to check that no file has been altered after the driver package has been published. To ensure the integrity of

[8] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/using-device-manager [Retrieved: 3/3/2023]

[9] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/components-of-a-driver-package [Retrieved: 3/3/2023]and https://learn.microsoft.com/en-us/windows-hardware/drivers/install/installation-component-overview [Retrieved: 3/3/2023]

[10] Microsoft provides an overview of the different section of INF file. See https://learn.microsoft.com/en-us/windows-hardware/drivers/install/summary-of-inf-sections for further information.

[11] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/inf-version-section [Retrieved: 3/3/2023]

the catalog file, it is digitally signed[12]. This file can be vendor generated or obtained after the driver package passed the tests of *Windows Hardware Quality Lab* (WHQL)[13].

- **Driver files**: A driver package may include one or more executable files, including the driver itself whose extension is `.sys` but also executable `.dll` files executing in kernel-mode, if needed.
- **Other files**: A driver package can also be seen as an archive which may include files which are user-mode executable or data files. For instance, one might have hardware calibration settings, Win32 services, device icons and so on…

The Driver Store serves as a trusted central repository for the resources provided by a driver package. Only resources located within the Driver Store can be used during the driver installation process. It must be emphasized that Microsoft considers two different operations related to the Driver Store and the driver installation process. The first operation, called staging, is the copy procedure of a driver package into the Driver Store. The second operation is called device installation, meaning a driver is fully deployed in the system and ready to run. A driver package must be staged to the Driver Store before it can be installed (and therefore used by any devices). Furthermore, it must be emphasized that if the device follows general hardware standards, there is no need (e.g., for a vendor) to stage a dedicated driver package. Indeed, Windows embeds in the Driver Store generic driver package resources for standard types or classes of devices. But in some cases, it may be relevant for a vendor to provide its own driver package. For instance, if there are no suitable driver package resources or better ones to provide enhanced functionality.

The Driver Store is located at `%SystemRoot%\System32\DriverStore`. This directory contains subdirectories for the individual driver package resources. Furthermore, the registry key `HKEY_LOCAL_MACHINE\SYSTEM\DriverDatabase\DriverPackages` is used to store additional information regarding driver packages and is associated with the Driver Store.

1.3.2 Device Instance

In the context of PnP, the term "device instance" refers to a specific occurrence of a particular device on a system. In technical terms, the PnP manager implemented within the kernel executive, maintains a critical kernel-mode data structure known as the device tree. During system startup, this data structure is initialized and continually updated each time a new device is inserted or removed from the system.

The device tree is composed of device nodes. The device tree keeps track of all the devices present in the system and is managed by the PnP. Therefore, a device node consists of all the instantiated information (i.e., data sets) that collectively form a device within the system. This includes the device objects that represent the device and other relevant information related to PnP, like configuration flags. In essence, a device instance refers to a device node that has been created within the device tree.

1.3.3 Windows Device Identification Strings

A device identification string in Windows is a unique identifier that can be assigned either by the manufacturer during the manufacturing process or by the operating system itself, typically during device installation or startup process. It usually consists of a combination of letters, numbers, and symbols that are specific to the device and its driver. There are several identification strings associated with a device. However, each of these identification strings, including the Device ID, Hardware IDs, Compatible IDs, Instance ID, and Container ID (starting from Windows 7), is unique and serves to identify a device in

[12] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/digital-signatures [Retrieved: 3/3/2023]

[13] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/whql-release-signature [Retrieved: 3/3/2023]

different contexts. Microsoft documents the device identification strings.[14] The following list provides a brief overview of important device identification strings:

- **Device ID** represents the device's device ID.[15] This string holds the most-specific possible description of the device. For instance, it includes the bus name, the manufacturer name, the device name, the revision number, the packager ID, and packaged product reference, whenever it is possible. However, the device ID is limited to a single kind of device, not a specific device of the system. The device ID string does not contain enough information to differentiate between two identical devices on the machine. For example, there could be two USB-sticks used for storage purpose and coming from the same manufacturer, with similar characteristics (same technology, same model, same storage capacity and so on). Both USB-sicks would have the same device ID.
- **Instance ID** represents the instance ID for the device.[16] It is used to differentiate between two identical devices (as discussed in the previous example). The instance ID is either unique across the whole computer or at least unique on the device's parent bus. That way, the format of the string is bus specific. The instance ID is persistent across system restart.
- **Hardware IDs** represent the IDs used to match the device and the driver in the system.[17] More directly, a Hardware ID is used to allow the device driver setup procedure to locate the most suitable driver package. This is a vendor-defined identification string that Windows uses to identify a suitable set of drivers to use with the device. Typically, each device instance has multiple Hardware IDs. The list of hardware IDs is ordered from the most-specific to most-general ID (the last one is usually identical to the device's device ID). This allows to match the most specific hardware ID with the associated driver package.
- **Compatible IDs** represent a list of IDs that can be used to link the device to a compatible driver package.[18] The format is the same than the one used with Hardware IDs, but the purpose of the compatible IDs list is to be a more generic description than the Hardware IDs. It does not include specific manufacturer or model information, but it represents the kind of device this hardware is. The same way as with hardware IDs, compatible IDs are listed in order of decreasing suitability.
- **Container ID** represents the device's container ID.[19] Typically, a device can be composed of sub-devices usually called functional devices or features. For instance, an antenna device may represent a Wi-Fi port or a Bluetooth port. In this case, the system sees two devices linked to different buses (Bluetooth and network). For many reasons, it may be convenient to regroup devices not per features but per physical devices (keeping the original tree architecture coming from the physical world). Starting with Windows 7, the container ID aims to regroup all functional devices from a single removable physical device. In a way, it identifies the physical device plugged into the system. That way, all functional devices from a physical device have the same container ID. The container ID is optional and depends on specific devices characteristics.

1.3.4 Unified Device Property Model

The Unified Device Property Model (UDPM) is a feature of the PnP Manager that provides a standard way to store and to retrieve information about device instances operated and maintained by the PnP Manager. In practice, it defines the representation of device properties[20] that describe the device instances and it provides a consistent way to access related information.

[14] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/device-identification-strings [Retrieved 3/3/2023]
[15] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/device-ids [Retrieved 3/3/2023]
[16] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/instance-ids [Retrieved 3/3/2023]
[17] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/hardware-ids [Retrieved 3/3/2023]
[18] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/compatible-ids [Retrieved 3/3/2023]
[19] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/container-ids [Retrieved 3/3/2023]
[20] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/device-properties [Retrieved: 3/3/2023]

A device property is data identified by a property key.[21] Formally speaking, a property key is a `DEVPROPKEY` structure[22], which is composed of two members. The first one is a property category represented as GUID, also named format ID (fmtid). It is used to reference a set of properties in a given category. A category represents common properties, usually sharing a common purpose or origin, which are defined by Microsoft or by the device's manufacturer. The second member is a property identifier (pid) used to uniquely identify the property within the property category. This is an unsigned value which (for internal system reasons) must be greater than or equal to two. In practice, a property key is usually defined thanks to the `DEFINE_DEVPROPKEY` macro[23] in source codes, allowing to provide a "string name" for the sake of convenience during the development (but this name is not kept, only the GUID and the value of the property identifier are kept).

```
DEFINE_DEVPROPKEY
In Windows Vista and later versions of Windows, the DEFINE_DEVPROPKEY macro creates a DEVPROPKEY
structure that represents a device property key in the unified device property model.

DEFINE_DEVPROPKEY(
  DEVPKEY_DeviceContainer_ConfigFlags,                                    //Friendly Name
  0x78c34fc8, 0x104a, 0x4aca, 0x9e, 0xa4, 0x52, 0x4d, 0x52, 0x99, 0x6e, 0x57,  //fmtid
  105);                                                                   //pid (0x65)
```

Code Block 1: Illustration of a device property definition.

Furthermore, each device property has a property-specific property-data-type identifier. More directly, the property-data-type identifier is a `DEVPROPTYPE`-typed value.[24] It is used to indicate which data type is associated with the returned value when a property is requested (e.g., integer, string, GUID, etc). The type of the value returned can be of fixed-length or variable-length. In addition, it is used to indicate if the value is single, composed as an array (collection of fixed-length base data types) or a list (collection of variable-length base data types) of values. For instance, the user-friendly name for the `DEVPKEY_DeviceContainer_ConfigFlags` device property's property-data-type identifier is `DEVPROP_TYPE_INT32`[25], which indicates that it returns a single fixed-length signed integer value.

The PnP manager stores device properties in a database. Specifically, the PnP manager stores and maintains device properties in the Windows registry, but not in a single registry key. Instead, it distributes properties over multiple registry keys. For example, some device properties are stored as subkeys of `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\`, where the subkey names are derived from the device instance ID. Similarly, the device container properties are stored in the device container registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceContainers\{Device Container GUID}\Properties`, each stored in a subkey named with the device container GUID (see Code Block 2).

```
!reg querykey \REGISTRY\MACHINE\SYSTEM\CONTROLSET001\CONTROL\DEVICECONTAINERS\{7d19918b-9b03-
11ed-87fa-0800276d26aa}\Properties\{78c34fc8-104a-4aca-9ea4-524d52996e57}\0065

Hive ffff8c0e1080c000
KeyNode ffff8c0e13a0607c

[ValueType] [ValueName] [ValueData]
FFFF0011 <nameless value> 0
```

Code Block 2: Example of a device property stored in the registry.

[21] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/property-keys [Retrieved: 3/3/2023]
[22] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/devpropkey [Retrieved: 3/3/2023]
[23] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/define-devpropkey [Retrieved: 3/3/2023]
[24] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/property-data-type-identifiers [Retrieved: 3/3/2023]
[25] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/devprop-type-int32 [Retrieved: 3/3/2023]

The registry keys are not supposed to be directly accessed by applications. To retrieve information, Microsoft provides two documented APIs to interact with the PnP Manager: The Setup API[26] and the Configuration Manager API[27]. On the one Hand, the Setup API is the older of the two, but still maintained for backward compatibility reasons. On the Other hand, the Configuration Manager API is the newer one that offers more advanced functionality than the Setup API. Therefore, Microsoft recommends using the Configuration Manager API for newer applications that require more advanced management capabilities. Despite the differences between the two APIs, they share common points when interacting with the PnP Manager. For example, both APIs can be used to read and write device properties.

1.3.5 Device Identification Procedure

The following section provides an overview of the steps involved in identifying a device when a USB stick is inserted into a Windows computer. This procedure is based on the original Microsoft documentation[28], and it will detail each step of the process to provide a better understanding of how device identification works in this context.

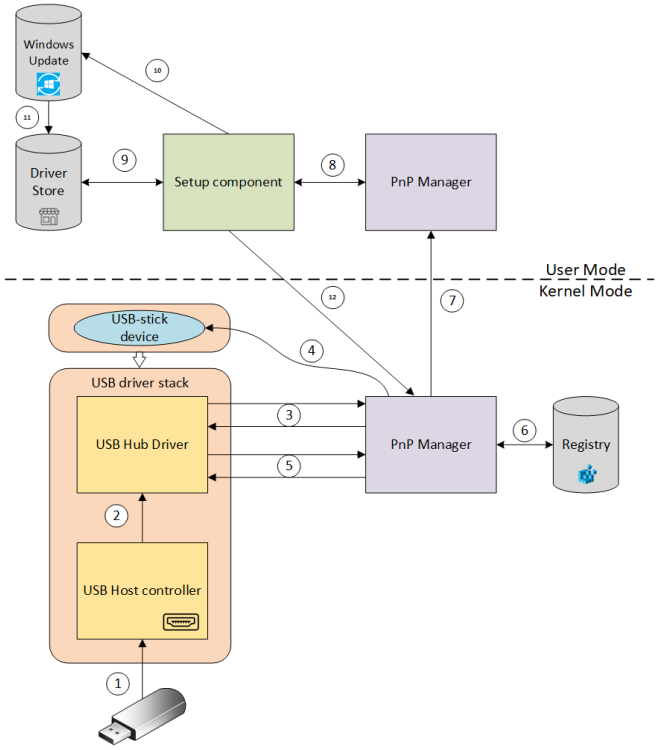

Figure 2: General overview of device driver identification procedure, leading to interface the Device Driver Manager.

1. When the USB host controller bus driver detects the presence of a USB stick several tasks are performed. For example, the USB host controller bus driver allocates the necessary hardware resources and propagates related events up (i.e., a "plug" event has happened) to the bus driver stack. This will ensure that the device is configured properly and will be ready to use.

[26] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/setupapi [Retrieved: 3/3/2023]
[27] https://learn.microsoft.com/en-us/mem/configmgr/develop/reference/configuration-manager-reference [Retrieved: 3/3/2023]
[28] https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/adding-a-pnp-device-to-a-running-system [Retrieved: 3/3/2023]

2. The hub driver recognizes the submitted "plug" event and determines that a new device is on its bus.
3. Next, the hub driver informs the PnP manager that a new device has arrived and that the device relationships must be evaluated. After that the PnP manager sends an `IRP_MN_QUERY_DEVICE_RELATIONS`[29] request to the hub driver. The Hub driver handles the request (i.e., if appropriate) and passes the request down to the next driver. In this case, the PnP manager is asking for the current list of devices present on the bus (bus relations). This procedure is called *bus enumeration*.[30]

   In this example, the USB hub driver handles the `IRP_MN_QUERY_DEVICE_RELATIONS` request. Therefore, it creates a Physical Device Object (PDO)[31] for the USB-stick device and includes a reference to the PDO in its list of devices returned within the `IRP_MN_QUERY_DEVICE_RELATIONS`.

4. Once the PnP manager received the current list of devices, it compares the list of bus relations returned in the `IRP_MN_QUERY_DEVICE_RELATIONS` with the current list stored in the PnP device tree data structure. If a new device is recognized, a device object structure is initialized for the USB-stick and linked as a node in the device tree managed by the PnP manager.
5. The PnP manager gathers information about the new device and begins configuring the device. This is done in a sequence of queries initiated by the PnP manager and resolved by the USB hub driver. In practice, it means the USB hub driver retrieves information from the device (by any means convenient to the bus technology) and translates it to be understandable by the PnP manager. Among the information retrieved, there are the device's IDs previously discussed but also the device's capacities (for instance, the device can be locked or ejected), bus information, device's needs for (hardware) resources, and an optional text description.
6. The PnP manager checks the device's configuration on the registry of Windows. In practice, if the device has already been seen in the system and it has been correctly configured, the registry key `HKLM\System\CurrentControlSet\Enum\<enumerator>\<deviceID>` should have been created. This key stores for each device its general configuration and all the IDs and different properties provided by the bus driver. It allows the PnP manager to set up the device's driver stack when the device will be reinserted another time.

For the remaining of the procedure, the device is considered as inserted for the first time and does not have a registry key.

7. The first time a device is inserted, a setup procedure is initiated for the device. Therefore, the kernel-mode PnP manager coordinates with the user-mode PnP manager to notify that there is a new device which must be installed. The notion of user-mode PnP manager is not defined by Microsoft, but in practice, it can be a service (see Section 2), such as `DsmSvc`.
8. In this scenario, the `DsmSvc` obtains information about the device to be installed and passes it on to a setup component responsible for carrying out the device setup procedure. To identify the appropriate driver packages needed for installation, the device's IDs are retrieved and used as a starting point.
9. During the setup procedure, the device's hardware IDs or compatible IDs are used to select the best driver package available in the Driver Store. However, driver packages can be installed through two methods: *software-first installation*, which happens before a hardware device is ever plugged in (for

[29] https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mn-query-device-relations [Retrieved: 3/3/2023]
[30] https://learn.microsoft.com/en-us/windows-hardware/drivers/wdf/creating-device-objects-in-a-bus-driver [Retrieved: 3/3/2023]
[31] https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-wdm-device-objects [Retrieved: 3/3/2023]

instance with a CDROM which would have been provided with the device), and *hardware-first installation*, which is triggered when a new hardware device is attached to the system.

10. If necessary (i.e., no suitable driver package is staged), the setup procedure may utilize network resources to locate the most appropriate driver package for the device. This can involve accessing Windows Update functionality, which in turn requests driver packages from Microsoft's servers.[32]
11. If the driver package is retrieved via Windows Update, it must be staged from the Windows Update temporary `SoftwareDistribution` folder to the driver store.
12. After the driver package has been successfully staged, it can be installed. Once installed, the setup procedure signals the kernel-mode PnP manager to load the newly installed drivers. Additionally, a notification may be sent to restart the device, which prompts the device's driver stack to be reevaluated and the installed drivers to be loaded.

[32] If no driver package can be located, the device remains in a non-functional state, which the user can see, for example, via the driver manager.

2 Technical Analysis of Functionalities

This section discusses relevant execution principles of the `Device Setup Manager` service, referred to as `DsmSvc`.[33] The `DsmSvc` service is part of the PnP infrastructure (see section 1.3.1), and its main purpose is to provide installation and configuration support for hardware devices and their drivers. It is implemented in the `%SystemRoot%\System32\DeviceSetupManager.dll` library file. It operates within a service host process (see `%SystemRoot%\system32\svchost.exe -k netsvcs -p`, in Code Block 3 *'BinaryPathName'*) and it is executed in the security context of the `LocalSystem` account, a predefined system account typically used for service operation[34][35].

```
PS C:\Users\ernw> Get-Service -Name DsmSvc | Select-Object DisplayName, Name, BinaryPathName,
UserName, StartupType, ServiceType

DisplayName    : Device Setup Manager
Name           : DsmSvc
BinaryPathName : C:\Windows\system32\svchost.exe -k netsvcs -p
UserName       : LocalSystem
StartupType    : Manual
ServiceType    : Win32OwnProcess, Win32ShareProcess
```

Code Block 3: Overview of the DsmSvc service.

However, for a better understanding, the following briefly discusses the core building blocks that make up the service. On the one hand, `DsmSvc` is basically a task scheduler that allows the PnP infrastructure to automate the installation and configuration of hardware devices and their drivers without the need for manual intervention. Therefore, it implements the following core building blocks to manage the execution of tasks efficiently:

- Thread Pool: A thread pool is a collection of worker threads used to execute tasks. When a task is ready to be executed, it is assigned to an available thread in the thread pool. This allows the task scheduler to execute multiple tasks concurrently, improving performance and reducing overall execution time (see section 2.2).
- Task Queue: When a task is scheduled, it is added to a queue. The queue manages the order in which tasks are executed and it ensures that they are executed in a timely manner (see section 2.3).
- Task Trigger: This is the condition that initiates the start of the task. The trigger is set to be run when a specific event occurs. In this case, when a device requiring support is plugged into the system (see sections 2.1 and 2.3).
- Task Actions: These are the actions that are performed when the task is triggered. For example, initializing the installation of a device driver (see section 2.4).

On the other hand, since `DsmSvc` is mainly a task scheduler integrated into a larger infrastructure (see section 1.3.1), it heavily relies on other components (e.g., to fetch data or to perform the actual work). In practice, this means that communication with other parts of the system plays a crucial role. For this purpose, the service uses a wide variety of technologies, such as the Unified Device Property Model (UDPM), to retrieve information about device instances (see section 1.3.4). Furthermore, as already mentioned, `DsmSvc` is responsible for supporting device instances that require installation and configuration of their drivers. In this context, the service is responsible for scheduling the underlying work. It needs to be emphasized that most of the scheduled work is not performed by the service itself. Indeed,

[33] (ERNW_WP2) discusses the general architecture of Windows, including Windows services. Debugging services is discusses in (ERNW_WP4).

[34] https://learn.microsoft.com/en-us/windows/win32/services/service-user-accounts?redirectedfrom=MSDN [Retrieved: 3/3/2023]

[35] This account has extensive privileges.

most of the heavy work is delegated to other components which are specifically designed for it. For example, delegating the download of a missing driver package to Windows Update.

2.1 Service Trigger

This section explains the circumstances under which the `DsmSvc` service starts. The operation related to services is managed by the system support process called "Service Control Manager" (SCM) (executable: `%SystemRoot%\System32\services.exe`). Among other things, SCM is responsible for starting, stopping, and interacting with services. Therefore, SCM creates an entry (referred to as "service record[36]") in a service database for each application that is registered as a managed service application. The registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services` is the nonvolatile representation of the services database, with the registry subkeys underneath representing individual service records. A service record contains all service-related parameters defined for a service to be managed accordingly by the SCM. For example, this entails the service's name, description, status, or startup type (i.e., manual, automatic, boot).

It is evident from the `DsmSvc` service record registry key, `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\DsmSvc`, that the service is registered to be started when a specific trigger event occurs. This means that the SCM does not automatically start the service when the system boots or the user logs in. Instead, it is only started by the SCM when it is needed. These triggers can be associated with various events. Among other things, this includes system activities, such as[37]:

- The system joins or leaves a domain.
- An event generated by a specific ETW provider occurs.
- A machine or user group policy change occurs.
- A message is pushed for a specific Windows Notification Facility (WNF) state name[38] (see also (Allievi, Ionescu, Russinovich, & Solomon, 2021) chapter 8 "Windows Notification Facility".

`DsmSvc` is a trigger-started service. This can be recognized from the startup type being set to manual (see `StartupType` in Code Block 3) coupled with the existence of a registry subkey named `TriggerInfo` under the `DsmSvc` service record registry key. This registry key only is present when a service is registered as a triggered service. It contains the individual trigger event meta parameters that were specified during service registration. However, since a service can be registered to be started by multiple events, the `TriggerInfo` registry key contains the individual trigger event meta parameters in a set of child keys named with indexes, starting from `0`. For example, `DsmSvc` is registered to listen for a single trigger event defined by the parameters stored under the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\DsmSvc\TriggerInfo\0`.

```
C:\Users\ernw>reg query
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\DsmSvc\TriggerInfo\0

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\DsmSvc\TriggerInfo\0
  Action      REG_DWORD   0x1
  Data0       REG_BINARY  7518BCA33D009602
              // as int   0x296003da3bc1875
  DataType0   REG_DWORD   0x1
  GUID        REG_BINARY  16287A2D5E0CFC459CE7570E5ECDE9C9
              // GUID     {2D7A2816-0C5E-45FC-9CE7-570E5ECDE9C9}
  Type        REG_DWORD   0x7
```

Code Block 4: DsmSvc trigger configuration HKLM\SYSTEM\CurrentControlSet\Services\DsmSvc\TriggerInfo\0.

[36] https://learn.microsoft.com/en-us/windows/win32/services/service-record-list [Retrieved 3/3/2023]
[37] A list of trigger types from which the activities can be derived can be found in section 4.1.
[38] A state name is usually represented by its "opaque value", which is a 64-bit unsigned integer.

Internally, SCM establishes and maintains an event system that ultimately ensures that SCM is notified to start a service when certain system activities occur. In practice, this means reading the trigger parameters from the `TriggerInfo` registry key (see Code Block 4) at system start and passing it to an event system. In practice, it allows SCM to subscribe to events generated by other components based on the trigger parameters and to receive notifications when they occur.

Code Block 5 shows multiple `WinDbg` statements, which ultimately represent the trigger parameters dealing with the start of `DsmSvc`. Based on type, subtype, and the trigger-specific data item, one can determine under which condition SCM is notified to start `DsmSvc`. In the context of `DsmSvc`, the `dwTriggerType` type field stores a `DWORD` value of `0x7` (`SERVICE_TRIGGER_TYPE_CUSTOM_SYSTEM_STATE_CHANGE`), while the `pTriggerSubtype` subtype field points to a Globally Unique Identifier (GUID) with a value of `{2D7A2816-0C5E-45FC-9CE7-570E5ECDE9C9}` (`CUSTOM_SYSTEM_STATE_CHANGE_EVENT_GUID`) (see Code Block 5 and Code Block 6).

It must be emphasized that in the case of custom trigger types (e.g., `SERVICE_TRIGGER_TYPE_CUSTOM_SYSTEM_STATE_CHANGE`), it is either an ETW or WNF-based trigger. For example, if the `dwTriggerType` field stores a `DWORD` value of `0x7`, it is a WNF and if it is `0x20` (`SERVICE_TRIGGER_TYPE_CUSTOM`), it means an ETW-based trigger. If it is of type `0x7`, the `pTriggerSubtype` field must also point to the GUID `2D7A2816-0C5E-45FC-9CE7-570E5ECDE9C9`. However, if it is of type `0x20`, the `pTriggerSubtype` field must refer to an ETW provider GUID.

Based on the definition of custom trigger types, the registered trigger for `DsmSvc` is based on WNF push message. The `pData` field points to an `unsigned __int64` value. In the Context of WNF, this value represents a WNF state name, which can be seen as the conceptually equivalent to an ETW event identifier or name[39].

```
0:000> dx -r3 @$SERVICE_TRIGGER_INFO = @$SyntheticTypes.CreateInstance("_SERVICE_TRIGGER_INFO",
0x000002b0a816fc50)
    cTriggers        : 0x1
    pTriggers        : * 0x2b0a816fcb0
        dwTriggerType    : 0x7
        dwAction         : 0x1
        pTriggerSubtype  : 0x2b0a816fd20 : {2D7A2816-0C5E-45FC-9CE7-570E5ECDE9C9} [Type: _GUID
*]
            [<Raw View>]     [Type: _GUID]
        cDataItems       : 0x1
        pDataItems       : * 0x2b0a816fd80
            dwDataType       : 0x1
            cbData           : 0x8
            pData            : 0x2b0a816fde0 : 0x75 [Type: unsigned char *]
    pReserved        : 0x0 [Type: unsigned char *]

0:000> dx @$pData = @$SERVICE_TRIGGER_INFO.pTriggers.pDataItems.pData

0:000> dx *(unsigned __int64*)@$pData
*(unsigned __int64*)@$pData : 0x296003da3bc1875 [Type: unsigned __int64]
```

Code Block 5: _SERVICE_TRIGGER_INFO showing the trigger type, subtype, and the associated data. In this case, the trigger type is WNF.

Code Block 6 shows a `WinDbg` statement of the WNF information stored in the `perf_nt_c.dll` file.[40] This information can be used to map the opaque WNF state name value to a human-readable name and description. In summary, the condition when SCM starts `DsmSvc` is when the WNF `0x296003da3bc1875` (`WNF_PNPC_CONTAINER_CONFIG_REQUESTED`) state name message is pushed. Based on the description in the `perf_nt_c.dll` file (see *'Description'*, in Code Block 6) this happens when a change requiring a device container (see section 1.3.3) to be updated occurred.

[39] https://learn.microsoft.com/en-us/windows-hardware/test/wpt/eventprovider [Retrieved 3/3/2023]
[40] The `perf_nt_c.dll` file (Windows Performance Analyzer) contains among other things information regarding WNF state names.

```
0:052> dx Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_WNF_STATE_NAME_TABLE",
0x00007ffacbf9F170)
    pStateName        : 0x7ffacc02ee48 : 0x296003da3bc1875 [Type: unsigned __int64 *]
    Name              : 0x7ffacc004a50 : "WNF_PNPC_CONTAINER_CONFIG_REQUESTED" [Type: wchar_t *]
    Description       : 0x7ffacc004aa0 : "A change that requires a device container to be updated
has occurred." [Type: wchar_t *]
```

Code Block 6: _WNF_STATE_NAME_TABLE with the discussed WNF state name, showing the state name and description.

This WNF message is triggered in the context of the kernel whenever the value of the `DEVPKEY_DeviceContainer_ConfigFlags` device property (see section 1.3.4) is set to `unconfiguredConfigFlags` (`0xFFFFFFFF`). As discussed in section 1.3.5 the kernel PnP manager is responsible to handle the first arrival of a device. In this context the `PiDcContainerRequiresConfiguration` function is responsible for issuing the WNF notification from the kernel which will trigger the start of `DsmSvc`.

Code Block 7 shows the pseudo-code of the `PiDcContainerRequiresConfiguration` function. Among others this function performs two tasks, first it sets device properties using the UDMP API, and secondly it pushes a WNF message. The latter is realized via the `ZwUpdateWnfStateData` function with the previously discussed `WNF_PNPC_CONTAINER_CONFIG_REQUESTED` state name as the first parameter. However, before this call the PnP manager updates the device property and sets `DEVPKEY_DeviceContainer_ConfigFlags` device property as discussed previously. This update relies on UDMP API which will be used thereafter by `DsmSvc` to retrieve information about the device to be installed. Indeed, in practice, there is no direct information provided to the service by a WNF notification.

```
PiDcContainerRequiresConfiguration(LPWSTR DeviceContainerId)
{
[...]
  NtStatus = PnpSetObjectProperty(
                       [...],
                       DeviceContainerId,                          //Device Container Id
                       [...],
                       &DEVPKEY_DeviceContainer_ConfigFlags,       //Property key
                       [...],
                       &unconfiguredConfigFlags,                   //Property value
                       [...]);
  if ( NtStatus >= 0 )
    ZwUpdateWnfStateData(
                StateName, // WNF_PNPC_CONTAINER_CONFIG_REQUESTED (0x296003DA3BC1875)
                [...]);
[...]
}
```

Code Block 7: The Plug and Play manager sets the value of the DeviceContainer_ConfigFlags device property to unconfiguredConfigFlags.

In summary, this section outlined under which circumstances `DsmSvc` is triggered and therefore started by SCM. This is the case when the PnP manager sets the `DeviceContainer_ConfigFlags` device property to `unconfiguredConfigFlags` (`0xFFFFFFFF`) and pushes the WNF message `CONTAINER_CONFIG_REQUESTED`. However, at this point, no information is transferred from the Plug and Play manager to the `DsmSvc` service about the device instance requiring support. The procedure only ensures that the service is started.

2.2 Service Initialization

This section provides an overview of the initial initialization of `DsmSvc`. On the one hand, initial service initialization includes general Windows service (e.g., common to all Windows services) initialization tasks which are discussed in detail in (Allievi, Ionescu, Russinovich, & Solomon, 2021) Chapter 10 “Windows

Services" and Microsoft's own documentation[41]. On the other hand, `DsmSvc` task scheduler specific initialization tasks, such as setup of the thread pool and task queue. Figure 3 shows the relevant functions involved during initialization.

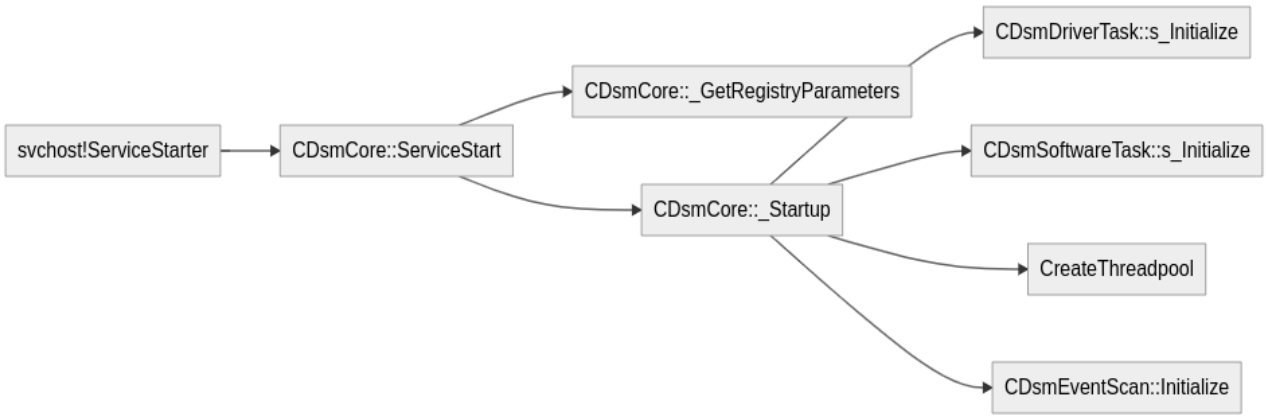

Figure 3: Call flow of DsmSvc initialization showing the important initialization steps.

First, important objects are created an instanciated. For example, `CDsmCore` (the core of `DsmSvc` task scheduler implementation) that holds relevant information of the task scheduler, such as information about the thread pool[42] and other important objects.

Second, the `CDsmCore::_GetRegistryParameters` function is called to read and evaluate the information stored in the registry under the `HKLM\Software\Microsoft\Windows\CurrentVersion\DeviceSetup` key. Among other things, this includes the registry values `MaxThreadCount` and `DebugBreakOnStart`. The first is used to set the maximum number of threads in the thread pool. The second is used to control the debugging behavior of `DsmSvc` (i.e., it performs a `__debugbreak`[43] directly after reading the value).

Next, the `CDsmCore::_Startup` function is called to initialize the thread pool and other important objects. In practice, this includes the following setup tasks:

- **Synchronization setup**: Since `DsmSvc` is a multithreaded service, synchronization must be taken care of. This includes locks (realized through `InitializeSRWLock`[44]) for the Driver-, Software- and Metadata tasks. Locks are either initialized directly inside `CDsmCore::_Startup` or in their respective initialization functions like `CDsmSoftwareTask::s_Initialize`.
- **Thread pool setup**: The thread pool of the service is initialized; this includes callbacks for workers (with `CDsmCore::_JobCallback`) and timers. This is done using the respective API functions. It first creates the thread pool by calling `CreateThreadpool`[45]. Next, the maximum number of worker threads is set via `SetThreadpoolThreadMaximum`[46]. Furthermore, the thread pool's timers are created using `CreateThreadpoolTimer`[47].

[41] https://learn.microsoft.com/en-us/windows/win32/services/service-servicemain-function [Retrieved 3/3/2023]
[42] https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pools [Retrieved 3/3/2023]
[43] https://learn.microsoft.com/en-us/cpp/intrinsics/debugbreak?view=msvc-170 [Retrieved 2/2/2023]
[44] https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-initializesrwlock [Retrieved 2/2/2023]
[45] https://learn.microsoft.com/en-us/windows/win32/api/threadpoolapiset/nf-threadpoolapiset-createthreadpool [Retrieved 2/2/2023]
[46] https://learn.microsoft.com/en-us/windows/win32/api/threadpoolapiset/nf-threadpoolapiset-setthreadpoolthreadmaximum [Retrieved 2/2/2023]
[47] https://learn.microsoft.com/en-us/windows/win32/api/threadpoolapiset/nf-threadpoolapiset-createthreadpooltimer [Retrieved 3/3/2023]

- **Job scheduler setup**: `CDsmCore::_Startup` calls `CDsmJobScheduler::Initialize`. This one sets a flag that the `CDsmJobScheduler` is initialized and, more importantly, calls `CDsmEventScan::Initialize`. This function is responsible for setting up callbacks regarding data retrieval from the Plug and Play manager and are discussed in the next section. This allows `DsmSvc` to obtain the information regarding which device instance requires support.

It must be emphasized, in general, drivers can be retrieved either from an online or offline source. This is decided during initialization in the `CDsmDriverTask::s_Initialize` function. This one checks the registry `HKLM\Software\Microsoft\Windows\CurrentVersion\DevicePath`[48] value. In practice this means it evaluates if the value is set to `%SystemRoot%\inf` (which is the default). If so then `CDsmDriverTask::s_fLocalSearchEnabled` is set to `0`, otherwise to `1`. `0` indicates that only an online search is performed and `1` that a locale search is performed before an online search.

During the course of the analysis of the version LTSC 2019, it has been observed that `CDsmDriverTask::s_fLocalSearchEnabled` is set to `0` by default (i.e., only an online search is performed). It must be mentioned that this does not correspond with the most recent documentation provided by Microsoft.[49] Code Block 8 shows the `CDsmDriverTask::s_Initialize` function of version LTSC 2019, which sets the value of `CDsmDriverTask::s_fLocalSearchEnabled` to `0` by default.[50]

However, in newer versions (i.e., 21H2), it was observed that the value is set to `1` in the default configuration. This means a locale search is performed before an online search which corresponds to the most recent documentation provided by Microsoft (see Code Block 9).

[48] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/preloading-driver-packages [3/3/2023]

[49] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/how-windows-selects-a-driver-for-a-device [3/3/2023]

[50] The file version is reported as follows: 10.0.17763.2989 (WinBuild.160101.0800)

```
void CDsmDriverTask::s_Initialize(void)
{
  LSTATUS LSTATUS;
  LSTATUS ERROR;
  DWORD pcbData;
  WCHAR lpDevicePathFromRegistry[20];

  InitializeSRWLock(&CDsmDriverTask::s_srwLock);
  memset_0(lpDevicePathFromRegistry, 0, 0x22ui64);
  pcbData = 0x22;
  LSTATUS = RegGetValueW(
              HKEY_LOCAL_MACHINE,
              L"Software\\Microsoft\\Windows\\CurrentVersion",
              L"DevicePath",
              0x10000004u,
              0i64,
              lpDevicePathFromRegistry,
              &pcbData);
  CDsmDriverTask::s_fLocalSearchEnabled = 0;
  if ( LSTATUS > 0 )                            // ERROR_SUCCESS
    ERROR = LSTATUS | 0x80070000;
  else
    ERROR = LSTATUS;
  if ( ERROR < 0 )
  {
    if ( LSTATUS != 2 )                         // ERROR_FILE_NOT_FOUND
    {
      if ( /* WPP Logging enabled */ )
        WPP_SF_D([...]);
      goto setLocalSearchEnabled;
    }
    [...]
  }
  else
  {
    if ( lpDevicePathFromRegistry[0] && lstrcmpiW(lpDevicePathFromRegistry,
L"%SystemRoot%\\inf") )
    {
      if ( /* WPP Logging enabled */ )
        WPP_SF_S([...]);
setLocalSearchEnabled:
      CDsmDriverTask::s_fLocalSearchEnabled = 1;
      return;
    }
    if ( /* WPP Logging enabled */ )
    {
      WPP_SF_([...]);
    }
  }
}
```

Code Block 8: CDsmDriverTask::s_Initialize in version LTSC 2019.

```
void CDsmDriverTask::s_Initialize(void)
{
  LSTATUS ERROR;
  LSTATUS _ERROR;
  LSTATUS __ERROR;
  DWORD pcbData;
  WCHAR lpDevicePathFromRegistry[17];

  InitializeSRWLock(&CDsmDriverTask::s_srwLock);
  pcbData = 34;
  memset(lpDevicePathFromRegistry, 0, sizeof(lpDevicePathFromRegistry));
  ERROR = RegGetValueW(
            HKEY_LOCAL_MACHINE,
            L"Software\\Microsoft\\Windows\\CurrentVersion",
            L"DevicePath",
            0x10000004u,
            0i64,
            lpDevicePathFromRegistry,
            &pcbData);
  CDsmDriverTask::s_fLocalSearchEnabled = 0;
  _ERROR = ERROR;
  if ( ERROR > 0 )                              // ERROR_SUCCESS
    __ERROR = ERROR | 0x80070000;
  else
    __ERROR = ERROR;
  if ( __ERROR >= 0 )
  {
    if ( !lpDevicePathFromRegistry[0] || !lstrcmpiW(lpDevicePathFromRegistry,
L"%SystemRoot%\\inf") )
    {
      /* WPP logging*/
    }
    if ( /* WPP logging enabled */ )
      WPP_SF_S([...]);
enableLocalSearch:
    CDsmDriverTask::s_fLocalSearchEnabled = 1;
    return;
  }
  if ( ERROR != 2 )                             // ERROR_FILE_NOT_FOUND
  {
    if (/* WPP logging enabled */ )
      WPP_SF_D([...]);
    goto enableLocalSearch;
  }
  /* WPP logging */
}
```

Code Block 9: CDsmDriverTask::s_Initialize in version 21H2.

In summary, this section outlined the initial initialization of the task scheduler implemented in `DsmSvc`. This among other things includes the creation and instantiation of important objects such as `CDsmCore`. Furthermore, data retrieval from the Plug and Play manager is configured. But at this point no data has been transmitted. The procedure only ensures that the task scheduler is setup and ready to perform the actual work.

2.3 Scheduler Queue Initialization

In a general way, the `DsmSvc` service is responsible to perform different task to install a device. In practice, the service can install many devices in a single launch. To proceed, it uses a complex procedure to organize the different tasks in an asynchronous way. In this section, the focus is on the initialization of the scheduler queue. Essentially, the task scheduler works by reading a list of device instance entries, each containing information about the jobs and tasks that it should perform. Figure 4 shows the conceptual structure of the scheduler queue. Each block represents a list of devices, jobs, or tasks in this figure.

- SchedulerQueue: The scheduler queue is a hash map that allows an efficient storage and retrieval of key-value pairs that associates a unique key (device container GUID) with a value (JobQueue).

- JobQueue: The job queue is a linked list of jobs that are waiting to be executed in order to support the new device.
- TaskQueue: The task queue is a linked list of predefined tasks to be executed in the context of a job.

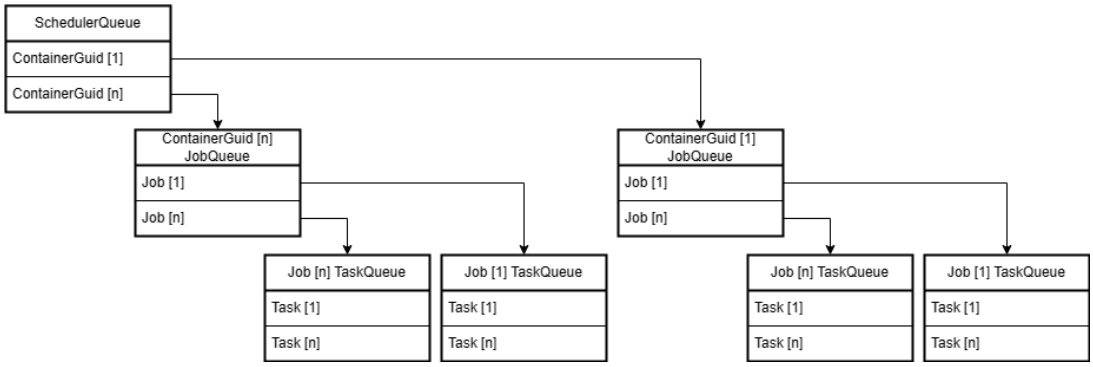

Figure 4: Overview of DmsSvc's scheduler queue

However, since the main purpose of `DsmSvc` is to provide support for device instances, in general, it matters to discuss the way the service retrieves the data that are related to a new device (see section 1.3.2). Windows provides the Unified Device Property Model (UDPM) API (see section 1.3.4) in order to store and to retrieve information about a device instances. This model defines the representation of properties for device instances and provides a consistent way to access information. For instance, it is possible to get access to the device's manufacturer, device type, or status, to name a few. As a side note, from a technical point of view, device properties can also be accessed either via the `CfgMgr32`[51] or the `SetupApi`[52] property functions.

At the beginning of the initialization of the tree-like scheduler queue structure, the device instances for which the start of `DsmSvc` has been triggered (i.e., device that needs support) must be obtained. The identification of which device instance is involved (i.e., represented by the corresponding device or device container GUID, see section 1.3.3) is obtained by querying the device property `DEVPKEY_DeviceContainer_ConfigFlags`. In this case, the device instances where the device property value has been updated (see section 2.1 and 1.3.4). A check if the value has been updated to `unconfiguredConfigFlags` is performed later in the callback function `CDsmEventScan::_DevQueryCallback`. Code Block 10 shows the pseudo-code of the UDPM request in the `DeviceSetupManager!CDsmEventScan::Initialize` function.

```
CDsmEventScan::Initialize([...])
{
  [...]
  DevCreateObjectQuery(
    DevObjectTypeDeviceContainer,          // Query filter (i.e. filter for object types)
    DevQueryFlagUpdateResults,             // Query filter (i.e. filter for object condition)
    [...]
    pRequestedProperties,                  // Property key
    [...]
    CDsmEventScan::_DevQueryCallback,      // Query result callback function
    [...]);
  [...]
}
```

Code Block 10: Pseudo code of CDsmEventScan::Initialize and its call to cfgmgr32!DevCreateObjectQuery function, with parameters querying properties in parameters.

The `cfgmgr32!DevCreateObjectQuery` function creates a device query for a device property, which means that the function is designed to retrieve data from a database based on a specific device property. The following parameters are of particular interest because they define the applied query:

[51] https://learn.microsoft.com/en-us/windows/win32/api/cfgmgr32/ [Retrieved 3/3/2023]
[52] https://learn.microsoft.com/en-us/windows/win32/api/setupapi/ [Retrieved 3/3/2023]

- The `DevObjectTypeDeviceContainer` parameter specifies a query filter. In this case, it defines that only device containers fall within the scope of the query.
- The `DevQueryFlagUpdateResults` parameter specifies a query filter, which defines that only properties flagged as updated fall within the scope of the query.
- The `pRequestedProperties` parameter specifies the property within the scope of the query. In this case `DEVPKEY_DeviceContainer_ConfigFlags`. Code Block 11 represents a `WinDbg` statement of the `pRequestedProperties` parameter of type `_DEVPROPCOMPKEY` (see Code Block 12).

```
0:004> dx -r2 Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_DEVPROPCOMPKEY", @r9),d
    Key
        fmtid            : {78C34FC8-104A-4ACA-9EA4-524D52996E57} [Type: _GUID]
        pid              : 105
    Store            : 0
```

Code Block 11: The 4th parameter (i.e. pRequestedProperties) passed to cfgmgr32!DevCreateObjectQuery.

```
DEFINE_DEVPROPKEY
In Windows Vista and later versions of Windows, the DEFINE_DEVPROPKEY macro creates a DEVPROPKEY
structure that represents a device property key in the unified device property model.

DEFINE_DEVPROPKEY(DEVPKEY_DeviceContainer_ConfigFlags, 0x78c34fc8, 0x104a, 0x4aca, 0x9e, 0xa4,
0x52, 0x4d, 0x52, 0x99, 0x6e, 0x57, 105);   // DEVPROP_TYPE_UINT32

propertyDescription
   shellPKey = DEVPKEY_DeviceContainer_ConfigFlags
   formatID = 78C34FC8-104A-4ACA-9EA4-524D52996E57
   propID = 105
   typeInfo
      type = UInt32
```

Code Block 12: DEVPKEY_DeviceContainer_ConfigFlags device property description.[53]

- The `DeviceSetupManager!CDsmEventScan::_DevQueryCallback` parameter specifies the callback function in which to process the queried results. Code Block 13 shows the prototype of the callback function.

```
typedef VOID (WINAPI *PDEV_QUERY_RESULT_CALLBACK)(
    _In_ HDEVQUERY hDevQuery,
    _In_opt_ PVOID pContext,
    _In_ const DEV_QUERY_RESULT_ACTION_DATA *pActionData
);
```

Code Block 13: Prototype of `CDsmEventScan::_DevQueryCallback`

As already mentioned, the `cfgmgr32!DevCreateObjectQuery` function does not return the result of the actual query. It just initializes the query. In practice, the result is processed in the supplied callback function when the query result is returned. As depicted in Code Block 13, the 3rd parameter (i.e., `pActionData` parameter of type `_DEV_QUERY_RESULT_ACTION_DATA`) references the data of the query result. Code Block 14 shows a `WinDbg` statement of the query result based on the previously discussed parameters. It shows:

- The updated property `fmtid: {78C34FC8-104A-4ACA-9EA4-524D52996E57}` highlighted in green.
- A reference (`Buffer: 0x208432ee3f8`) to the updated value `0xffffffff` (corresponding to `unconfiguredConfigFlags`, see section 2.1 Code Block 7) highlighted in yellow.
- The device container GUID `pszObjectId: {7d19918b-9b03-11ed-87fa-0800276d26aa}` highlighted in cyan. This is the device whose queried property was updated. In this context, it means the device instance that requires support.

[53] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/define-devpropkey [Retrieved 3/3/2023]

```
0:006> dx -r6
Debugger.Utility.Analysis.SyntheticTypes.CreateInstance("_DEV_QUERY_RESULT_ACTION_DATA", @r8)
    Action           : 0x1
    Data
        DeviceObject
            ObjectType       : 0x2
            pszObjectId      : 0x208432ee378 : "{7d19918b-9b03-11ed-87fa-0800276d26aa}" [Type:
wchar_t *]
            cPropertyCount   : 0x1
            pProperties      : * 0x208432ee3c8
                CompKey
                    Key
                        fmtid            : {78C34FC8-104A-4ACA-9EA4-524D52996E57} [Type: _GUID]
                        pid              : 0x69
                    Store            : 0x0
                Type             : 0x7
                BufferSize       : 0x4
                Buffer           : 0x208432ee3f8 [Type: void *]

0:006> dx (unsigned int*)0x208432ee3f8
(unsigned int*)0x208432ee3f8                 : 0x208432ee3f8 : 0xffffffff [Type: unsigned int *]
    0xffffffff [Type: unsigned int]
```

Code Block 14: Example `_DEV_QUERY_RESULT_ACTION_DATA` passed to `CDsmEventScan::_DevQueryCallback`

Since the device instances for which the `DsmSvc` start was triggered (i.e., represented by the corresponding device container GUID) are now known, the actual initiation of the scheduler queue can now be carried out. Code Block 15 shows the pseudo-code of the implementation `CDsmJobScheduler::_PostJob` concerning the initiation of the scheduler queue.

```
CDsmJobScheduler::_PostJob(
        CDsmJobScheduler *pJobScheduler,
        const enum DSMJOB_REQUEST_TYPE *const DsmJobRequestTypes,
        unsigned int Count,
        [...],
        const struct _GUID *pGuid,
        [...])
{
  [...]
  do
  {
      [...]
      CDsmJobScheduler::_CreateDeviceJobQueue(pJobScheduler, pGuid, pDeviceJobQueue);
      [...]
      CDsmJobDescription::CDsmJobDescription(pJobDescription, [...], DsmJobRequestType, [...]);
      [...]
      CDeviceJobQueue::AddJob(pDeviceJobQueue, pJobDescription)
      [...]
      CDsmJobScheduler::_AddDeviceToPendingQueue(pJobScheduler, pGuid)
      [...]
      CDsmCore::CreateJobWorker([...]);
      [...]
      --Count;
  } while (Count)
  [...]
}
```

Code Block 15: Pseudo code of `CDsmJobScheduler::_PostJob`.

The scheduler queue is initialized in multiple steps, starting after the service retieved the device container GUIDs that requier support. First, the `CDsmJobScheduler::_CreateDeviceJobQueue` function is called. Among other things, the main purpose of this function is to create an instance of the `CDeviceJobQueue` object and to add it as value to a `ATL::CAtlMap`[54] map object with the device container GUID as its key. A map object is a hash table with a system of keys and values.

[54] https://learn.microsoft.com/en-us/cpp/atl/reference/catlmap-class?view=msvc-170 [Retrieved 3/3/2023]

Second, the `CDsmJobDescription::CDsmJobDescription` function is called once a `CDeviceJobQueue::CDeviceJobQueue` object exists. The purpose of the `CDsmJobDescription::CDsmJobDescription` function is to initialize the data members of the `CDsmJobDescription` object, which essentially defines the job to be performed. This includes information such as the GUID of the device container (i.e., `pGuid`) and the requested job, which is specified by the `DsmJobRequestType` integer value.

It must be mentioned that jobs are defined on the basis of the `DsmJobRequestTypes` enumerator. Each integer value (i.e., `DsmJobRequestType`) of the enumerator defines a specific job type. At this point, the job type does not state what task it consists of. In addition, it must also be mentioned that job types are not defined based on any particular logic. In practice, this means the `DsmJobRequestTypes` enumerator consists of four integer values 1-4. Each of these values defines a particular job type, and all available job types are always executed one after the other in the order of 1-4.

Third, the function `CDeviceJobQueue::AddJob` is called after the `CDsmJobDescription` object has been generated. This function adds the previously generated `CDsmJobDescription` object to an `ATL::CAtlList`[55] list object referenced by the `pDeviceJobQueue` (see *JobQueue* in Figure 4).

Fourth, through the `CDsmJobScheduler::_AddDeviceToPendingQueue` function, the device container GUID is added to a `ATL::CAtlList` list object referenced by the `pJobScheduler`. This list is not part of the schedule queue itself. Instead, it is used in parallel to keep track of which device container GUIDs are still in progress.

Finally, the `CDsmCore::CreateJobWorker` function calls the worker thread `CDsmCore::_JobCallback` callback function (see section 2.2). This one calls `CDsmJobScheduler::RunNextJob` to run a scheduled job. In practice, this means, on the one hand, pop the entry (i.e., the `CDsmJobDescription` object) from the beginning of the `ATL::CAtlList` list object referenced by the `pDeviceJobQueue` (see *JobQueue* in Figure 4). On the other hand, create an instance of the `CDsmJob` object and copy the information from the `CDsmJobDescription` object into it before calling the `CDsmJob::RunJob` function with the `CDsmJob` as its first parameter. This function is where the actual job work is carried out.

As already mentioned, the job type referenced by the `CDsmJobDescription` does not state what tasks it consists of. For this reason, the actual tasks that are carried out in the context of a specific job type must now finally be defined. Therefore, the `CDsmJob::_InitializeTaskQueue` function is first called by `CDsmJob::RunJob`. This function uses the job type to determine what tasks make up a job. Each individual task is represented as an individual integer value between 1-10. Job types and task values are in a one-to-many relationship. That is, a job consists of one or more tasks. However, the exact composition is determined via an *if chain*. After the integer values of the tasks have been determined, they are added one by one to an `ATL::CAtlList` list object referenced by the `CDsmJob` object (see *TaskQueue* in Figure 4). After the function has been successfully executed, the entire queue of task for a given job is setup, and the work can begin. Figure 5 shows in a matrix how exactly the jobs and tasks are related and their assigned integer values. Note that some tasks and jobs may not be related, possibly due to legacy or future features, but are still present in the code.

[55] https://learn.microsoft.com/en-us/cpp/atl/reference/catllist-class?view=msvc-170 [Retrieved 3/3/2023]

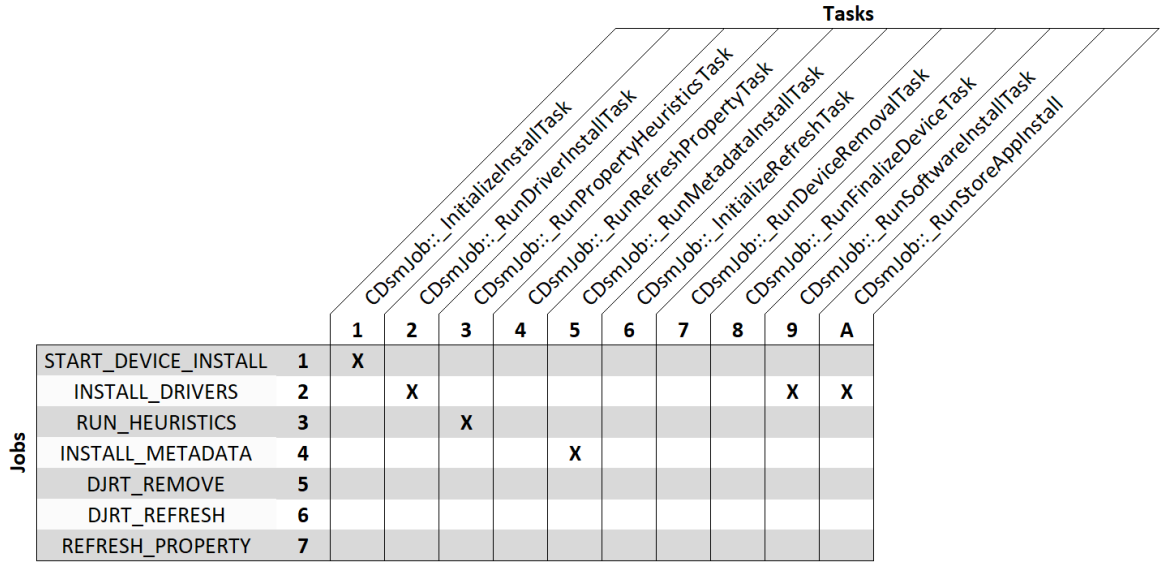

Figure 5: Relationship between jobs and tasks.

In summary this section outlined the initialization of the scheduler queue. In addition, it describes the relationship between jobs and tasks. The procedure only ensures the tree-like queue structure is setup. This means, that each job and its coresspondings tasks are defined and ready to be executed. However, up to this point, no task is executed.

2.4 Online Retrieval and Staging Procedure

As discussed in the previous section, the support of a device instance consists of multiple jobs and their tasks. However, the focus of this section is the exemplary discussion of the `CDsmJob::_RunDriverInstallTask` function (i.e., task see Figure 5).

To have a device driver perfectly operational, it must be emphasized there are two conceptual operational steps to proceed: staging and installing (see section 1.3.1.8). In short, staging means making the driver packages available on the system (i.e., in the Driver Store). Installation means deployment of the driver package so device drivers and other dependencies can be used to interface the device. To avoid confusion, symbol names referenced in this section (i.e., `InstallBestPackages`) imply installation. However, installation in this context finally means staging.

The `CDsmJob::_RunDriverInstallTask` function performs the retrieval and staging of a driver package (see section 1.3.1.8). As already mentioned, up to this point, only the device container GUID is known. Furthermore, drivers can be retrieved either from an online or an offline source. The decision on this will be made during the initial initialization of the task scheduler, as a general switch for the whole service (see section 2.2). However, since the online source is used by default in the discussed case, this case will now be discussed in more detail below.

In order to retrieve and to stage driver packages, the function `CDsmJob::_RunDriverInstallTask` first needs to perform some preparation. This includes, among other things, identifying the device instance ID (for example `USB\VID_8087&PID_0026\5&18F54CB7&0&2` (see section 1.3.3)) of the device grouped under the device container GUID that requires support. Once completed, the actual staging procedure is performed by the `CDsmDriverTask::_DownloadAndInstallDriver` function. This function mainly uses the device driver retrieval client API (DDRC) implemented in the `DeviceDriverRetrievalClient.dll` file to retrieve and stage driver packages. As mentioned, an online source is used by default to retrieve and stage driver packages. In this case, the functionality provided by DDRC is mainly based on the documented functionality provided by the Windows Update Agent API (WUA) implemented in `wuapi.dll` file. Figure 6 shows a condensed Call Graph of

`CDsmDriverTask::_DownloadAndInstallDriver`, highlighting the functions involved when a driver package is retrieved and staged.

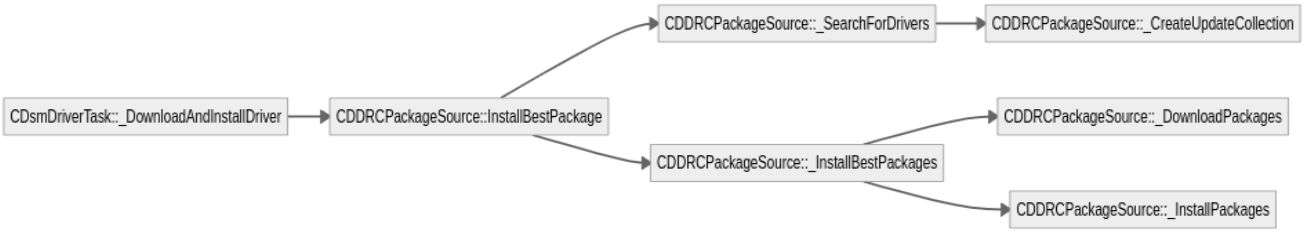

Figure 6: Condensed Call Graph of CDsmDriverTask::_DownloadAndInstallDriver function prefixed with CDDRCPackageSource are implemented in DeviceDriverRetrivalClient.dll

To retrieve and stage a driver package for the previously identified hardware ID, the `CDDRCPackageSource::InstallBestPackage` function is called. As previously mentioned, in the context where the driver is retrieved online, WUA will be heavily used internally. However, the work performed by this function can conceptually be divided into four steps:

- Search for driver packages (implemented in `CDDRCPackageSource::_SearchForDrivers`);
- Select the appropriate driver package (implemented in `CDDRCPackageSource::_InstallBestPackages`);
- Download the selected driver package (implemented in `CDDRCPackageSource::_DownloadPackages`);
- and finally, stage the downloaded driver package (implemented in `CDDRCPackageSource::_InstallPackages`).

Search for driver packages: The search for a suitable driver package is initiated by the `CDDRCPackageSource::_SearchForDrivers` function.

This function first checks the relevant configuration settings regarding online search. Therefore, the two registry values `SearchOrderConfig` and `DriverServerSelection` stored under the registry keys `HKLM\Software\Microsoft\Windows\CurrentVersion\DriverSearching` and `HKLM\Software\Policies\Microsoft\Windows\DriverSearching` are evaluated.

- `SearchOrderConfig`: this value specifies the search order. The value can be set via the group policy ` *Specify search order for device driver source locations* ` discussed in section 3.1. A value of `1` has been observed, corresponding to the default configuration, which means, in the end, an online search will be performed.
- `DriverServerSelection`: this value specifies if Windows Update or a managed server (Windows Server Update Services (WSUS)[56]) should be used. The value can be set via the group policy `*Specify the search server for device driver updates*`, discussed in section 3.1. During the cause of the analysis, it has been observed that the value is not set. Therefore, the default value of 0 is used, which means only Windows Update is enabled.

In order to perform a Windows Update driven search the function `CDDRCPackageSource::_CreateCollectionFromWU` is called. This function calls `CDDRCPackageSource::_CreateUpdateCollection` to create an `UpdateCollection`[57] object.

[56] https://learn.microsoft.com/en-us/windows-server/administration/windows-server-update-services/get-started/windows-server-update-services-wsus [Retrieved 3/3/2023]

[57] https://learn.microsoft.com/en-us/previous-versions/windows/desktop/bb313515(v=vs.85) [Retrieved 3/3/2023]

This object will contain the available updates provided by Microsoft once it will be fully initialized. Therefore, the function is called with the following parameters:

1. A pointer to `CDDRCPackageSource` struct.
2. An indicator of whether Windows Update or WSUS should be used.
3. An output parameter for the created `UpdateCollection`.

Internally, in order to create an `UpdateCollection` object, the `CDDRCPackageSource::_CreateUpdateCollection` function performs the following work. First, the client application's name of the `UpdateCollection` is set using `put_ClientApplicationID`[58] (in this case "`Device Driver Retrieval Client`"). Next, an asynchronous search is triggered based on a search criteria using the `wuapi!CUpdateSearcher::BeginSearch`[59] function. `DsmSvc` will wait (using `WaitForMultipleObjects`) until the search is completed and it triggers the end of the search by calling the `wuapi!CUpdateSearcher::EndSearch`[60] function. This function finally returns a collection of driver packages (i.e., `SearchResult`[61] objects) that match the former search criteria. Code Block 16 contains a `WinDbg` statement showing the `BeginSearch` function's search criteria. The criteria parameter is hardcoded and, in every case, set to "`CallerProfile='AU' and IsInstalled=0 and Type='Driver'`".[62]

```
(102c.18fc): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: DF74:30
wuapi!CUpdateSearcher::BeginSearch:
0x7ffbe0a98fa0 4055            push    rbp
||0:0:010> dx (BSTR) @rdx
(BSTR) @rdx                 : 0x2084330bb68 : "CallerProfile='AU' and IsInstalled=0 and
Type='Driver'" [Type: BSTR]
    67 'C' [Type: wchar_t]
```

Code Block 16: Criteria for Update search

Once the `SearchResult` is available, the `get_Updates` function is called to obtain the actual `UpdateCollection` from the search result. This `UpdateCollection` is then returned in the third parameter of the `CDDRCPackageSource::_CreateUpdateCollection` function.

Select the appropriate driver package: Once an `UpdateCollection` has been successfully retrieved, an evaluation is made to know which driver package in the collection fits best the device needing support. Therefore, the device's hardware ID and the compatibility ID are obtained through the `CDDRCPackageSource::InstallBestPackage` function. Code Block 17 shows `WinDbg` output of the retrieved hardware and compatibility IDs. These IDs are retrieved through `cfgmgr32!DevGetObjectProperties` (see section 1.3.4 for further details) wrapped in the function `GetDeviceObjectStringList`. In practice, the result is provided as a `PROPVARIANT`[63] datatype, containing the hardware or compatibility ID (see Code Block 17 and Code Block 18).

[58] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdatesearcher-put_clientapplicationid [Retrieved 3/3/2023]
[59] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdatesearcher-beginsearch [Retrieved 3/3/2023]
[60] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdatesearcher-endsearch [Retrieved 3/3/2023]
[61] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nn-wuapi-isearchresult [Retrieved 3/3/2023]
[62] Part of the criteria is presented in the documentation of the Search method (https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdatesearcher-search). The type parameter in the request specifies if a driver or software is searched. In addition, only updates not already installed are searched.
[63] https://learn.microsoft.com/en-us/windows/win32/api/propidlbase/ns-propidlbase-propvariant [Retrieved 3/3/2023]

```
||0:0:010> dx -r2 (PROPVARIANT*) 0xdaa89ff150
(PROPVARIANT*) 0xdaa89ff150                : 0xdaa89ff150 : vector of LPWSTR = {size = 0x2}
[Type: PROPVARIANT *]
    [<Raw View>]     [Type: PROPVARIANT]
    LPWSTR           : {size = 0x2} [Type: tagCALPWSTR]
        [<Raw View>]     [Type: tagCALPWSTR]
        [size]           : 0x2 [Type: unsigned long]
        [0]              : 0x2084332ae90 : "USB\VID_8087&PID_0026&REV_0002" [Type: wchar_t *]
        [1]              : 0x20843334790 : "USB\VID_8087&PID_0026" [Type: wchar_t *]
    vt               : 0x101f [Type: unsigned short]
```

Code Block 17: Retrieval of hardware ID

```
||0:0:010> dx -r2 (PROPVARIANT*) 0xdaa89ff168
(PROPVARIANT*) 0xdaa89ff168                : 0xdaa89ff168 : vector of LPWSTR = {size = 0x3}
[Type: PROPVARIANT *]
    [<Raw View>]     [Type: PROPVARIANT]
    LPWSTR           : {size = 0x3} [Type: tagCALPWSTR]
        [<Raw View>]     [Type: tagCALPWSTR]
        [size]           : 0x3 [Type: unsigned long]
        [0]              : 0x2084332aa80 : "USB\Class_E0&SubClass_01&Prot_01" [Type: wchar_t *]
        [1]              : 0x208433341d0 : "USB\Class_E0&SubClass_01" [Type: wchar_t *]
        [2]              : 0x20843326ae0 : "USB\Class_E0" [Type: wchar_t *]
    vt               : 0x101f [Type: unsigned short]
```

Code Block 18: Retrieval of compatible ID

After the hardware and the compatibility IDs are available, the `CDDRCPackageSource::_InstallBestPackages` function is called. This function determines which driver package referenced by the `UpdateCollection` fits the best. In this function, each driver package is iterated and retrieved using the `get_Item`[64] function. Afterwards, for each driver package, the `get_DriverHardwareID`[65] function is called to retrieve the hardware ID, or the compatible ID associated with the package. Code Block 19 shows a `WinDbg` statement with the hardware ID retrieved from the function.

```
DeviceDriverRetrievalClient!CDDRCPackageSource::_InstallBestPackages+0x133:
00000208`43e03db7 85c0            test    eax,eax
||0:0:010> dx  (BSTR*) 0x000000daa89ff070
(BSTR*) 0x000000daa89ff070                 : 0xdaa89ff070 [Type: BSTR *]
    0x208432e7bc8 : "usb\vid_8087&pid_0026&rev_0002" [Type: BSTR]
```

Code Block 19: Hardware ID of the software update

Finally, the IDs associated with the device instance and the driver package are compared using the `lstrcmpiW`[66] function. The driver package where either the hardware or compatible ID matches the associated device instance IDs is considered for download.

Download the selected driver package: The driver package identified in the previous step is now downloaded using the `CDDRCPackageSource::_DownloadPackages` function. This function initiates the `UpdateDownloader`[67] as defined in the registry. Once initiated, an asynchronous download is

[64] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdatecollection-get_item [Retrieved 3/3/2023]

[65] https://learn.microsoft.com/bs-latn-ba/windows/win32/api/wuapi/nf-wuapi-iwindowsdriverupdateentry-get_driverhardwareid [Retrieved 3/3/2023]

[66] https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-lstrcmpiw [Retrieved 3/3/2023]

[67] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nn-wuapi-iupdatedownloader [Retrieved 3/3/2023]

performed using the `BeginDownload`[68] and `EndDownload`[69] functions. Ultimately, this means that the driver package identified is downloaded to the `C:\Windows\SoftwareDistribution` directory.

Staging the downloaded driver package: Finally, the downloaded package is staged using the `CDDRCPackageSource::_InstallPackages` function. Just like the download, the staging is performed asynchronously using the `BeginInstall`[70] and `EndInstall`[71] functions. In practice, staging means calling up the `drvint.exe` program in a dedicated process. Among other things, it means copying the downloaded files (e.g., inf file see section 1.3.1.8) to the Driver Store located under the `%windir%\System32\DriverStore` directory and writing information in the registry key `HKLM\DRIVERS\DriverDatabase`. If this step is successful, the driver package is considered staged and the `CDsmJob::_RunDriverInstallTask` task is now complete.

In summary, this section outlined the retrieval and staging of a driver package. Therefore, Windows Update is used to perform the actual work. The procedure only ensures that the driver package is staged, however this does not mean that the driver package is installed and directly runnable (see section 1.3.1.8).

[68] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdatedownloader-begindownload [Retrieved 3/3/2023]
[69] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdatedownloader-enddownload [Retrieved 3/3/2023]
[70] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdateinstaller-begininstall [Retrieved 3/3/2023]
[71] https://learn.microsoft.com/en-us/windows/win32/api/wuapi/nf-wuapi-iupdateinstaller-endinstall [Retrieved 3/3/2023]

3 Configuration and Logging Capabilities

3.1 Configuration Capabilities

This section discusses the group policies settings and the written help available directly in Windows, which can be used for configuring device installation behavior. The settings are located at the policy path `Computer Configuration` → `Administrative Templates` → `System` → `Device Installation` and `User Configuration` → `Administrative Templates` → `System` → `Device Installation`. The following provides descriptions about the group policy settings and the registry implementation of the settings located at this path[72]:

3.1.1 DeviceInstallation.admx

Prioritize all digitally signed drivers equally during the driver ranking and selection process
Description (as provided by Microsoft): This policy setting allows you to determine how drivers signed by a Microsoft Windows Publisher certificate are ranked with drivers signed by other valid Authenticode signatures during the driver selection and installation process. Regardless of this policy setting, a signed driver is still preferred over a driver that is not signed at all.
If you enable or do not configure this policy setting, drivers that are signed by a Microsoft Windows Publisher certificate and drivers that are signed by other Authenticode certificates are prioritized equally during the driver selection process. Selection is based on other criteria, such as version number or when the driver was created.
If you disable this policy setting, drivers that are signed by a Microsoft Windows Publisher certificate are selected for installation over drivers that are signed by other Authenticode certificates.
Registry implementation: Table 2 presents the registry implementation of the policy setting "*Prioritize all digitally signed drivers equally during the driver ranking and selection process*".

Name	Value
Registry hive	`HKEY_LOCAL_MACHINE`
Registry path	`Software\Policies\Microsoft\Windows\DeviceInstall\Settings`
Value name	`AllSigningEqual`
Value type	`REG_DWORD`
Enabled value	`1`
Disabled value	`0`

Table 2: Registry implementation of "Prioritize all digitally signed drivers equally during the driver ranking and selection process"

Configure device installation time-out
Description (as provided by Microsoft): This policy setting allows you to configure the number of seconds Windows waits for a device installation task to complete.
If you enable this policy setting, Windows waits for the number of seconds you specify before terminating the installation.
If you disable or do not configure this policy setting, Windows waits 240 seconds for a device installation task to complete before terminating the installation.
Registry implementation: Table 3 presents the registry implementation of the policy setting "*Configure device installation time-out*".

Name	Value
Registry hive	`HKEY_LOCAL_MACHINE`
Registry path	`Software\Policies\Microsoft\Windows\DeviceInstall\Settings`
Value name	`InstallTimeout`

[72] The description is taken from the group policy files.

Value type	REG_DWORD

Table 3: Registry implementation of "Configure device installation time-out"

Prevent creation of a system restore point during device activity that would normally prompt creation of a restore point

Description (as provided by Microsoft): This policy setting allows you to prevent Windows from creating a system restore point during device activity that would normally prompt Windows to create a system restore point. Windows normally creates restore points for certain driver activity, such as the installation of an unsigned driver. A system restore point enables you to more easily restore your system to its state before the activity.

If you enable this policy setting, Windows does not create a system restore point when one would normally be created.

If you disable or do not configure this policy setting, Windows creates a system restore point as it normally would.

Registry implementation: Table 4 presents the registry implementation of the policy setting "*Prevent creation of a system restore point during device activity that would normally prompt creation of a restore point*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Settings
Value name	DisableSystemRestore
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 4: Registry implementation of "Prevent creation of a system restore point during device activity that would normally prompt creation of a restore point"

Allow remote access to the Plug and Play interface

Description (as provided by Microsoft): This policy setting allows you to allow or deny remote access to the Plug and Play interface.

If you enable this policy setting, remote connections to the Plug and Play interface are allowed.

If you disable or do not configure this policy setting, remote connections to the Plug and Play interface are not allowed.

Registry implementation: Table 5 presents the registry implementation of the policy setting "*Allow remote access to the Plug and Play interface*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Settings
Value name	AllowRemoteRPC
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 5: Registry implementation of "Allow remote access to the Plug and Play interface"

Allow administrators to override Device Installation Restriction policies

Description (as provided by Microsoft): This policy setting allows you to determine whether members of the Administrators group can install and update the drivers for any device, regardless of other policy settings.

If you enable this policy setting, members of the Administrators group can use the Add Hardware wizard or the Update Driver wizard to install and update the drivers for any device. If you enable this policy setting

on a remote desktop server, the policy setting affects redirection of the specified devices from a remote desktop client to the remote desktop server.
If you disable or do not configure this policy setting, members of the Administrators group are subject to all policy settings that restrict device installation.
Registry implementation: Table 6 presents the registry implementation of the policy setting "*Allow administrators to override Device Installation Restriction policies*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	AllowAdminInstall
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 6: Registry implementation of "Allow administrators to override Device Installation Restriction policies"

Allow installation of devices using drivers that match these device setup classes
Description (as provided by Microsoft): This policy setting allows you to specify a list of device setup class globally unique identifiers (GUIDs) for driver packages that Windows is allowed to install. This policy setting is intended to be used only when the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting is enabled, however it may also be used with the "Prevent installation of devices not described by other policy settings" policy setting for legacy policy definitions.
When this policy setting is enabled together with the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting, Windows is allowed to install or update driver packages whose device setup class GUIDs appear in the list you create, unless another policy setting at the same or higher layer in the hierarchy specifically prevents that installation, such as the following policy settings:
- Prevent installation of devices for these device classes
- Prevent installation of devices that match these device IDs
- Prevent installation of devices that match any of these device instance IDs
If the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting is not enabled with this policy setting, then any other policy settings specifically preventing installation will take precedence.
NOTE: The "Prevent installation of devices not described by other policy settings" policy setting has been replaced by the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting for supported target Windows 10 versions. It is recommended that you use the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting when possible.
Alternatively, if this policy setting is enabled together with the "Prevent installation of devices not described by other policy settings" policy setting, Windows is allowed to install or update driver packages whose device setup class GUIDs appear in the list you create, unless another policy setting specifically prevents installation (for example, the "Prevent installation of devices that match these device IDs" policy setting, the "Prevent installation of devices for these device classes" policy setting, the "Prevent installation of devices that match any of these device instance IDs" policy setting, or the "Prevent installation of removable devices" policy setting).
If you enable this policy setting on a remote desktop server, the policy setting affects redirection of the specified devices from a remote desktop client to the remote desktop server.
If you disable or do not configure this policy setting, and no other policy setting describes the device, the "Prevent installation of devices not described by other policy settings" policy setting determines whether the device can be installed.
Registry implementation: Table 7 presents the registry implementation of the policy setting "*Allow installation of devices using drivers that match these device setup classes*".

Name	Value

Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	AllowDeviceClasses
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 7: Registry implementation of "Allow installation of devices using drivers that match these device setup classes"

Prevent installation of devices using drivers that match these device setup classes

Description (as provided by Microsoft): This policy setting allows you to specify a list of device setup class globally unique identifiers (GUIDs) for driver packages that Windows is prevented from installing. By default, this policy setting takes precedence over any other policy setting that allows Windows to install a device.

NOTE: To enable the "Allow installation of devices that match any of these device IDs" and "Allow installation of devices that match any of these device instance IDs" policy settings to supersede this policy setting for applicable devices, enable the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting.

If you enable this policy setting, Windows is prevented from installing or updating driver packages whose device setup class GUIDs appear in the list you create. If you enable this policy setting on a remote desktop server, the policy setting affects redirection of the specified devices from a remote desktop client to the remote desktop server.

If you disable or do not configure this policy setting, Windows can install and update devices as allowed or prevented by other policy settings.

Registry implementation: Table 8 presents the registry implementation of the policy setting "*Prevent installation of devices using drivers that match these device setup classes*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	DenyDeviceClasses
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 8: Registry implementation of "Prevent installation of devices using drivers that match these device setup classes"

Allow installation of devices that match any of these device IDs

Description (as provided by Microsoft): This policy setting allows you to specify a list of Plug and Play hardware IDs and compatible IDs for devices that Windows is allowed to install. This policy setting is intended to be used only when the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting is enabled, however it may also be used with the "Prevent installation of devices not described by other policy settings" policy setting for legacy policy definitions.

When this policy setting is enabled together with the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting, Windows is allowed to install or update any device whose Plug and Play hardware ID or compatible ID appears in the list you create, unless another policy setting at the same or higher layer in the hierarchy specifically prevents that installation, such as the following policy settings:

- Prevent installation of devices that match these device IDs
- Prevent installation of devices that match any of these device instance IDs

If the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting is not enabled with this policy setting, then any other policy settings specifically preventing installation will take precedence.
NOTE: The "Prevent installation of devices not described by other policy settings" policy setting has been replaced by the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting for supported target Windows 10 versions. It is recommended that you use the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting when possible.
Alternatively, if this policy setting is enabled together with the "Prevent installation of devices not described by other policy settings" policy setting, Windows is allowed to install or update any device whose Plug and Play hardware ID or compatible ID appears in the list you create, unless another policy setting specifically prevents that installation (for example, the "Prevent installation of devices that match any of these device IDs" policy setting, the "Prevent installation of devices for these device classes" policy setting, the "Prevent installation of devices that match any of these device instance IDs" policy setting, or the "Prevent installation of removable devices" policy setting).
If you enable this policy setting on a remote desktop server, the policy setting affects redirection of the specified devices from a remote desktop client to the remote desktop server.
If you disable or do not configure this policy setting, and no other policy setting describes the device, the "Prevent installation of devices not described by other policy settings" policy setting determines whether the device can be installed.
Registry implementation: Table 9 presents the registry implementation of the policy setting "*Allow installation of devices that match any of these device IDs*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	AllowDeviceIDs
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 9: Registry implementation of "Allow installation of devices that match any of these device IDs"

Prevent installation of devices that match any of these device IDs
Description (as provided by Microsoft): This policy setting allows you to specify a list of Plug and Play hardware IDs and compatible IDs for devices that Windows is prevented from installing. By default, this policy setting takes precedence over any other policy setting that allows Windows to install a device.
NOTE: To enable the "Allow installation of devices that match any of these device instance IDs" policy setting to supersede this policy setting for applicable devices, enable the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting.
If you enable this policy setting, Windows is prevented from installing a device whose hardware ID or compatible ID appears in the list you create. If you enable this policy setting on a remote desktop server, the policy setting affects redirection of the specified devices from a remote desktop client to the remote desktop server.
If you disable or do not configure this policy setting, devices can be installed and updated as allowed or prevented by other policy settings.
Registry implementation: Table 10 presents the registry implementation of the policy setting "*Prevent installation of devices that match any of these device IDs*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	DenyDeviceIDs
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 10: Registry implementation of "Prevent installation of devices that match any of these device IDs"

Allow installation of devices that match any of these device instance IDs

Description (as provided by Microsoft): This policy setting allows you to specify a list of Plug and Play device instance IDs for devices that Windows is allowed to install. This policy setting is intended to be used only when the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting is enabled, however it may also be used with the "Prevent installation of devices not described by other policy settings" policy setting for legacy policy definitions.
When this policy setting is enabled together with the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting, Windows is allowed to install or update any device whose Plug and Play device instance ID appears in the list you create, unless another policy setting at the same or higher layer in the hierarchy specifically prevents that installation, such as the following policy settings:
- Prevent installation of devices that match any of these device instance IDs
If the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting is not enabled with this policy setting, then any other policy settings specifically preventing installation will take precedence.
NOTE: The "Prevent installation of devices not described by other policy settings" policy setting has been replaced by the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting for supported target Windows 10 versions. It is recommended that you use the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting when possible.
Alternatively, if this policy setting is enabled together with the "Prevent installation of devices not described by other policy settings" policy setting, Windows is allowed to install or update any device whose Plug and Play device instance ID appears in the list you create, unless another policy setting specifically prevents that installation (for example, the "Prevent installation of devices that match any of these device IDs" policy setting, the "Prevent installation of devices for these device classes" policy setting, the "Prevent installation of devices that match any of these device instance IDs" policy setting, or the "Prevent installation of removable devices" policy setting).
If you enable this policy setting on a remote desktop server, the policy setting affects redirection of the specified devices from a remote desktop client to the remote desktop server.
If you disable or do not configure this policy setting, and no other policy setting describes the device, the "Prevent installation of devices not described by other policy settings" policy setting determines whether the device can be installed.

Registry implementation: Table 11 presents the registry implementation of the policy setting "*Allow installation of devices that match any of these device instance IDs*".

Name	Value
Registry hive	`HKEY_LOCAL_MACHINE`
Registry path	`Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions`
Value name	`AllowInstanceIDs`
Value type	`REG_DWORD`
Enabled value	`1`
Disabled value	`0`

Table 11: Registry implementation of "Allow installation of devices that match any of these device instance IDs"

Prevent installation of devices that match any of these device instance IDs
Description (as provided by Microsoft): This policy setting allows you to specify a list of Plug and Play device instance IDs for devices that Windows is prevented from installing. This policy setting takes precedence over any other policy setting that allows Windows to install a device.
If you enable this policy setting, Windows is prevented from installing a device whose device instance ID appears in the list you create. If you enable this policy setting on a remote desktop server, the policy setting affects redirection of the specified devices from a remote desktop client to the remote desktop server.
If you disable or do not configure this policy setting, devices can be installed and updated as allowed or prevented by other policy settings.
Registry implementation: Table 12 presents the registry implementation of the policy setting "*Prevent installation of devices that match any of these device instance IDs*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	DenyInstanceIDs
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 12: Registry implementation of "Prevent installation of devices that match any of these device instance IDs"

Prevent installation of removable devices
Description (as provided by Microsoft): This policy setting allows you to prevent Windows from installing removable devices. A device is considered removable when the driver for the device to which it is connected indicates that the device is removable. For example, a Universal Serial Bus (USB) device is reported to be removable by the drivers for the USB hub to which the device is connected. By default, this policy setting takes precedence over any other policy setting that allows Windows to install a device.
NOTE: To enable the "Allow installation of devices using drivers that match these device setup classes", "Allow installation of devices that match any of these device IDs", and "Allow installation of devices that match any of these device instance IDs" policy settings to supersede this policy setting for applicable devices, enable the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting.
If you enable this policy setting, Windows is prevented from installing removable devices and existing removable devices cannot have their drivers updated. If you enable this policy setting on a remote desktop server, the policy setting affects redirection of removable devices from a remote desktop client to the remote desktop server.
If you disable or do not configure this policy setting, Windows can install and update driver packages for removable devices as allowed or prevented by other policy settings.
Registry implementation: Table 13 presents the registry implementation of the policy setting "*Prevent installation of removable devices*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	DenyRemovableDevices
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 13: Registry implementation of "Prevent installation of removable devices"

Prevent installation of devices not described by other policy settings

Description (as provided by Microsoft): This policy setting allows you to prevent the installation of devices that are not specifically described by any other policy setting.
NOTE: This policy setting has been replaced by the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting to provide more granular control. It is recommended that you use the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting instead of this policy setting.
If you enable this policy setting, Windows is prevented from installing or updating the driver package for any device that is not described by either the "Allow installation of devices that match any of these device IDs", the "Allow installation of devices for these device classes", or the "Allow installation of devices that match any of these device instance IDs" policy setting.
If you disable or do not configure this policy setting, Windows is allowed to install or update the driver package for any device that is not described by the "Prevent installation of devices that match any of these device IDs", "Prevent installation of devices for these device classes" policy setting, "Prevent installation of devices that match any of these device instance IDs", or "Prevent installation of removable devices" policy setting.

Registry implementation: Table 14 presents the registry implementation of the policy setting "*Prevent installation of devices not described by other policy settings*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	DenyUnspecified
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 14: Registry implementation of "Prevent installation of devices not described by other policy settings"

Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria

Description (as provided by Microsoft): This policy setting will change the evaluation order in which Allow and Prevent policy settings are applied when more than one install policy setting is applicable for a given device. Enable this policy setting to ensure that overlapping device match criteria is applied based on an established hierarchy where more specific match criteria supersedes less specific match criteria. The hierarchical order of evaluation for policy settings that specify device match criteria is as follows:
Device instance IDs > Device IDs > Device setup class > Removable devices
Device instance IDs
1. Prevent installation of devices using drivers that match these device instance IDs
2. Allow installation of devices using drivers that match these device instance IDs
Device IDs
3. Prevent installation of devices using drivers that match these device IDs
4. Allow installation of devices using drivers that match these device IDs
Device setup class
5. Prevent installation of devices using drivers that match these device setup classes
6. Allow installation of devices using drivers that match these device setup classes
Removable devices
7. Prevent installation of removable devices
NOTE: This policy setting provides more granular control than the "Prevent installation of devices not described by other policy settings" policy setting. If these conflicting policy settings are enabled at the same time, the "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria" policy setting will be enabled and the other policy setting will be ignored.
If you disable or do not configure this policy setting, the default evaluation is used. By default, all "Prevent installation..." policy settings have precedence over any other policy setting that allows Windows to install a device.

Registry implementation: Table 15 presents the registry implementation of the policy setting "*Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	AllowDenyLayered
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 15: Registry implementation of "Apply layered order of evaluation for Allow and Prevent device installation policies across all device match criteria"

Time (in seconds) to force reboot when required for policy changes to take effect

Description (as provided by Microsoft): This policy setting establishes the amount of time (in seconds) that the system will wait to reboot in order to enforce a change in device installation restriction policies.
If you enable this policy setting, set the amount of seconds you want the system to wait until a reboot.
If you disable or do not configure this policy setting, the system does not force a reboot.
Note: If no reboot is forced, the device installation restriction right will not take effect until the system is restarted.

Registry implementation: Table 16 presents the registry implementation of the policy setting "*Time (in seconds) to force reboot when required for policy changes to take effect*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions
Value name	ForceReboot
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 16: Registry implementation of "Time (in seconds) to force reboot when required for policy changes to take effect"

Display a custom message title when device installation is prevented by a policy setting

Description (as provided by Microsoft): This policy setting allows you to display a custom message title in a notification when a device installation is attempted and a policy setting prevents the installation.
If you enable this policy setting, Windows displays the text you type in the Main Text box as the title text of a notification when a policy setting prevents device installation.
If you disable or do not configure this policy setting, Windows displays a default title in a notification when a policy setting prevents device installation.

Registry implementation: Table 17 presents the registry implementation of the policy setting "*Display a custom message title when device installation is prevented by a policy setting*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions\DeniedPolicy
Value name	SimpleText
Value type	REG_SZ

Table 17: Registry implementation of "Display a custom message title when device installation is prevented by a policy setting"

Display a custom message when installation is prevented by a policy setting

Description (as provided by Microsoft): This policy setting allows you to display a custom message to users in a notification when a device installation is attempted and a policy setting prevents the installation.
If you enable this policy setting, Windows displays the text you type in the Detail Text box when a policy setting prevents device installation.
If you disable or do not configure this policy setting, Windows displays a default message when a policy setting prevents device installation.
Registry implementation: Table 18 presents the registry implementation of the policy setting "*Display a custom message when installation is prevented by a policy setting*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Restrictions\DeniedPolicy
Value name	DetailText
Value type	REG_SZ

Table 18: Registry implementation of "Display a custom message when installation is prevented by a policy setting"

Allow non-administrators to install drivers for these device setup classes
Description (as provided by Microsoft): This policy setting specifies a list of device setup class GUIDs describing driver packages that non-administrator members of the built-in Users group may install on the system.
If you enable this policy setting, members of the Users group may install new drivers for the specified device setup classes. The drivers must be signed according to Windows Driver Signing Policy, or be signed by publishers already in the TrustedPublisher store.
If you disable or do not configure this policy setting, only members of the Administrators group are allowed to install new driver packages on the system.
Registry implementation: Table 19 presents the registry implementation of the policy setting "*Allow non-administrators to install drivers for these device setup classes*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DriverInstall\Restrictions
Value name	AllowUserDeviceClasses
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 19: Registry implementation of "Allow non-administrators to install drivers for these device setup classes"

Code signing for driver packages
Description (as provided by Microsoft): Determines how the system responds when a user tries to install driver package files that are not digitally signed.
This setting establishes the least secure response permitted on the systems of users in the group. Users can use System in Control Panel to select a more secure setting, but when this setting is enabled, the system does not implement any setting less secure than the one the setting established.
When you enable this setting, use the drop-down box to specify the desired response.
-- "Ignore" directs the system to proceed with the installation even if it includes unsigned files.
-- "Warn" notifies the user that files are not digitally signed and lets the user decide whether to stop or to proceed with the installation and whether to permit unsigned files to be installed. "Warn" is the default.
-- "Block" directs the system to refuse to install unsigned files. As a result, the installation stops, and none of the files in the driver package are installed.
To change driver file security without specifying a setting, use System in Control Panel. Right-click My Computer, click Properties, click the Hardware tab, and then click the Driver Signing button.

Registry implementation: Table 20 presents the registry implementation of the policy setting "*Code signing for driver packages*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows NT\Driver Signing
Value name	BehaviorOnFailedVerify
Value type	REG_DWORD
Values	0 - Ignone 1 - Warn 2 - Block

Table 20: Registry implementation of "Code signing for driver packages"

3.1.2 DeviceSetup.admx

Turn off "Found New Hardware" balloons during device installation
Description (as provided by Microsoft): This policy setting allows you to turn off "Found New Hardware" balloons during device installation.
If you enable this policy setting, "Found New Hardware" balloons do not appear while a device is being installed.
If you disable or do not configure this policy setting, "Found New Hardware" balloons appear while a device is being installed, unless the driver for the device suppresses the balloons.
Registry implementation: Table 21 presents the registry implementation of the policy setting "*Turn off "Found New Hardware" balloons during device installation*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Settings
Value name	DisableBalloonTips
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 21: Registry implementation of "Turn off "Found New Hardware" balloons during device installation"

Do not send a Windows error report when a generic driver is installed on a device
Description (as provided by Microsoft): Windows has a feature that sends "generic-driver-installed" reports through the Windows Error Reporting infrastructure. This policy allows you to disable the feature.
If you enable this policy setting, an error report is not sent when a generic driver is installed.
If you disable or do not configure this policy setting, an error report is sent when a generic driver is installed.
Registry implementation: Table 22 presents the registry implementation of the policy setting "*Do not send a Windows error report when a generic driver is installed on a device*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Settings
Value name	DisableSendGenericDriverNotFoundToWER
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 22: Registry implementation of "Do not send a Windows error report when a generic driver is installed on a device"

Prevent Windows from sending an error report when a device driver requests additional software during installation

Description (as provided by Microsoft): Windows has a feature that allows a device driver to request additional software through the Windows Error Reporting infrastructure. This policy allows you to disable the feature.

If you enable this policy setting, Windows will not send an error report to request additional software even if this is specified by the device driver.

If you disable or do not configure this policy setting, Windows sends an error report when a device driver that requests additional software is installed.

Registry implementation: Table 23 presents the registry implementation of the policy setting "*Prevent Windows from sending an error report when a device driver requests additional software during installation*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DeviceInstall\Settings
Value name	DisableSendRequestAdditionalSoftwareToWER
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 23: Registry implementation of "Prevent Windows from sending an error report when a device driver requests additional software during installation"

Configure driver search locations

Description (as provided by Microsoft): This setting configures the location that Windows searches for drivers when a new piece of hardware is found.

By default, Windows searches the following places for drivers: local installation, floppy drives, CD-ROM drives, Windows Update.

Using this setting, you may remove the floppy and CD-ROM drives from the search algorithm.

If you enable this setting, you can remove the locations by selecting the associated check box beside the location name.

If you disable or do not configure this setting, Windows searches the installation location, floppy drives, and CD-ROM drives.

Note: To prevent searching Windows Update for drivers also see "Turn off Windows Update device driver searching" in Administrative Templates/System/Internet Communication Management/Internet Communication settings.

Registry implementation: Table 24 presents the registry implementation of the policy setting "*Configure driver search locations*".

Name	Value
Registry hive	HKEY_CURRENT_USER
Registry path	Software\Policies\Microsoft\Windows\DriverSearching
Value names	DontSearchFloppies DontSearchCD DontSearchWindowsUpdate
Value type	REG_DWORD
Enabled value	1

Table 24: Registry implementation of "Configure driver search locations"

Turn off Windows Update device driver search prompt

Description (as provided by Microsoft): Specifies whether the administrator will be prompted about going to Windows Update to search for device drivers using the Internet.

Note: This setting only has effect if "Turn off Windows Update device driver searching" in "Administrative Templates/System/Internet Communication Management/Internet Communication settings" is disabled or not configured.

If you enable this setting, administrators will not be prompted to search Windows Update.

If you disable or do not configure this setting, and "Turn off Windows Update device driver searching" is disabled or not configured, the administrator will be prompted for consent before going to Windows Update to search for device drivers.
Registry implementation: Table 25 presents the registry implementation of the policy setting "*Turn off Windows Update device driver search prompt*".

Name	Value
Registry hive	HKEY_CURRENT_USER
Registry path	Software\Policies\Microsoft\Windows\DriverSearching
Value name	DontPromptForWindowsUpdate
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 25: Registry implementation of "Turn off Windows Update device driver search prompt"

Turn off Windows Update device driver search prompt
Description (as provided by Microsoft): Specifies whether the administrator will be prompted about going to Windows Update to search for device drivers using the Internet.
Note: This setting only has effect if "Turn off Windows Update device driver searching" in "Administrative Templates/System/Internet Communication Management/Internet Communication settings" is disabled or not configured.
If you enable this setting, administrators will not be prompted to search Windows Update.
If you disable or do not configure this setting, and "Turn off Windows Update device driver searching" is disabled or not configured, the administrator will be prompted for consent before going to Windows Update to search for device drivers.
Registry implementation: Table 26presents the registry implementation of the policy setting "*Turn off Windows Update device driver search prompt*".

Name	Value
Registry hive	HKEY_CURRENT_USER
Registry path	Software\Policies\Microsoft\Windows\DriverSearching
Value name	DontPromptForWindowsUpdate
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 26: Registry implementation of "Turn off Windows Update device driver search prompt"

Specify search order for device driver source locations
Description (as provided by Microsoft): This policy setting allows you to specify the order in which Windows searches source locations for device drivers.
If you enable this policy setting, you can select whether Windows searches for drivers on Windows Update unconditionally, only if necessary, or not at all.
Note that searching always implies that Windows will attempt to search Windows Update exactly one time. With this setting, Windows will not continually search for updates. This setting is used to ensure that the best software will be found for the device, even if the network is temporarily available.
If the setting for searching only if needed is specified, then Windows will search for a driver only if a driver is not locally available on the system.
If you disable or do not configure this policy setting, members of the Administrators group can determine the priority order in which Windows searches source locations for device drivers.
Registry implementation: Table 27 presents the registry implementation of the policy setting "*Specify search order for device driver source locations*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DriverSearching
Value name	SearchOrderConfig
Value type	REG_DWORD

Values	0 – Always search Windows Update 1 – Search Windows Update only if needed 2 - Do not search Windows Update

Table 27: Registry implementation of "Specify search order for device driver source locations"

Specify the search server for device driver updates
Description (as provided by Microsoft): This policy setting allows you to specify the search server that Windows uses to find updates for device drivers.
If you enable this policy setting, you can select whether Windows searches Windows Update (WU), searches a Managed Server, or a combination of both.
Note that if both are specified, then Windows will first search the Managed Server, such as a Windows Server Update Services (WSUS) server. Only if no update is found will Windows then also search Windows Update.
If you disable or do not configure this policy setting, members of the Administrators group can determine the server used in the search for device drivers.
Registry implementation: Table 28 presents the registry implementation of the policy setting "*Specify the search server for device driver updates*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	Software\Policies\Microsoft\Windows\DriverSearching
Value name	DriverServerSelection
Value type	REG_DOWRD
Values	0 – Search Windows Update 1 – Search Managed Server 2 - Search Managed Server, then WU

Table 28: Registry implementation of "Specify the search server for device driver updates"

Prevent device metadata retrieval from the Internet
Description (as provided by Microsoft): This policy setting allows you to prevent Windows from retrieving device metadata from the Internet.
If you enable this policy setting, Windows does not retrieve device metadata for installed devices from the Internet. This policy setting overrides the setting in the Device Installation Settings dialog box (Control Panel > System and Security > System > Advanced System Settings > Hardware tab).
If you disable or do not configure this policy setting, the setting in the Device Installation Settings dialog box controls whether Windows retrieves device metadata from the Internet.
Registry implementation: Table 29 presents the registry implementation of the policy setting "*Prevent device metadata retrieval from the Internet*".

Name	Value
Registry hive	HKEY_LOCAL_MACHINE
Registry path	SOFTWARE\Policies\Microsoft\Windows\Device Metadata
Value name	PreventDeviceMetadataFromNetwork
Value type	REG_DWORD
Enabled value	1
Disabled value	0

Table 29: Registry implementation of "Prevent device metadata retrieval from the Internet"

3.2 Logging Capabilities

ETW can be used to record activities related to DsmSvc. Table 30 shows the manifest based and Trace Logging Providers ETW providers implemented in DsmSvc. The table in the Appendix, section Event IDs, present the Event IDs, and their descriptions, under which the manifest based ETW provider Microsoft-

`Windows-DeviceSetupManager` may record events. Furthermore, Appendix, Code Block 21 shows the relevant PowerShell command to retrieve the information.

ETW Provider	GUID
`Microsoft-Windows-DeviceSetupManager`	`FCBB06BB-6A2A-46E3-ABAA-246CB4E508B2`
`Microsoft.Windows.DeviceSetupManager`	`AB11A476-79F6-5026-7D54-2E9B9E539A2D`

Table 30: ETW providers of DsmSvc

Events from `Microsoft-Windows-DeviceSetupManager` are also used in the Windows Event Log under `Application and Service Logs` → `Microsoft` → `Windows` → `DeviceSetupManager`. The respective files are `C:\Windows\System32\winevt\Log\Msicrosoft-Windows-DeviceSetupManager%4Admin.evtx` and `C:\Windows\System32\winevt\Log\Microsoft-Windows-DeviceSetupManager%4Operational.evtx`.

4 Appendix

4.1 List of Trigger Types

```
// Service trigger types
//
#define SERVICE_TRIGGER_TYPE_DEVICE_INTERFACE_ARRIVAL               1
#define SERVICE_TRIGGER_TYPE_IP_ADDRESS_AVAILABILITY                2
#define SERVICE_TRIGGER_TYPE_DOMAIN_JOIN                            3
#define SERVICE_TRIGGER_TYPE_FIREWALL_PORT_EVENT                    4
#define SERVICE_TRIGGER_TYPE_GROUP_POLICY                           5
#define SERVICE_TRIGGER_TYPE_NETWORK_ENDPOINT                       6
#define SERVICE_TRIGGER_TYPE_CUSTOM_SYSTEM_STATE_CHANGE             7
#define SERVICE_TRIGGER_TYPE_CUSTOM                                 20
```

Code Block 20: List of trigger types

4.2 Event IDs

Code Block 21 shows a PowerShell script for retrieving event IDs and description if the ETW provider `Microsoft-Windows-DeviceSetupManager`. The results are listed in Table 31.

```
$event = Get-WinEvent -ListProvider "Microsoft-Windows-DeviceSetupManager"
$event[0].Events | Select-Object Id, Template, Description | ConvertTo-Csv
```

Code Block 21: Retrieving event metadata for ETW provider Microsoft-Windows-DeviceSetupManager

Event ID	Template	Description
100	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_CoreServiceMode" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_Event_Window_Seconds" inType="win:Int64" outType="xs:long"/> </template>	DSM service start, mode is %1, last session (or boot) was %2 seconds ago
101	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_UpTime_Seconds" inType="win:Int64" outType="xs:long"/> <data name="Prop_WorkTime_MilliSeconds" inType="win:Int64" outType="xs:long"/> </template>	DSM Service shutting down. Service uptime was %1 seconds, active worktime was %2 MilliSeconds.
102		DSM Service dll has loaded.
103		DSM Service dll is unloading.
104	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	DSM Service failed to start, result=%1
105		DSM Service is entering a retry sequence because soft (retryable) errors were encountered

106	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_RetryCycleCount" inType="win:UInt32" outType="xs:unsignedInt"/> </template>	DSM Service is leaving the retry state, there have been %1 retry cycles in this session
108	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_CoreServiceState" inType="win:UInt32" outType="xs:unsignedInt"/> </template>	The DSM service has entered service state '%1'
109	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_CoreServiceMode" inType="win:UInt32" outType="xs:unsignedInt"/> </template>	The DSM service has entered service mode '%1'
110	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> <data name="Prop_JobId" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_JobType" inType="win:UInt32" outType="xs:unsignedInt"/> </template>	Job (%2) has started for device container '%1', type=%3
111	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> <data name="Prop_JobId" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_JobStatus" inType="win:UInt32" outType="xs:unsignedInt"/> </template>	Job (%2) has completed for device container '%1', status=%3
112	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DeviceName" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> <data name="Prop_TaskCount" inType="win:Int32" outType="xs:int"/> <data name="Prop_PropertyCount" inType="win:Int32" outType="xs:int"/> <data name="Prop_WorkTime_MilliSeconds" inType="win:Int64" outType="xs:long"/> </template>	Device '%1' (%2) has been serviced, processed %3 tasks, wrote %4 properties, active worktime was %5 milliseconds.
120	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_MilliSeconds" inType="win:Int64" outType="xs:long"/> </template>	Driver update %1 has been downloaded from Windows Update, download time was %2 milliseconds

121	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	Driver install failed, result=%2 for devnode '%1'
122		Access to drivers on Windows Update was blocked by policy
123	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_Seconds" inType="win:Int32" outType="xs:int"/> <data name="Prop_DeviceId" inType="win:UnicodeString" outType="xs:string"/> </template>	The DSM service was delayed by %1 seconds for a driver query/download/install on device '%2'
124	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_DeviceInstanceId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_MilliSeconds" inType="win:Int64" outType="xs:long"/> </template>	Driver %1 was installed on device %2, install time was %3 milliseconds
125	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> </template>	Installation of a driver on device %1 was blocked by PnP restriction policy
126	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DeviceInstanceId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	Device '%1' matched driver update %2
130	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_MetadataPackageId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> <data name="Prop_StageTimeMilliSeconds" inType="win:Int64" outType="xs:long"/> </template>	Metadata package %1 has been staged for container %2, time was %3 milliseconds
131	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	Metadata staging failed, result=%2 for container '%1'

150	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DeviceName" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> </template>	The device '%1' with container ID %2, has been removed.
151	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DeviceName" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> </template>	The device '%1' with container ID %2, failed to respond to a device remove request.
152	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	Removal of device node '%1' failed with error code %2.
160	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwareName" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_DeviceInstanceId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_InstallTime" inType="win:UInt64" outType="xs:unsignedLong"/> </template>	Software %1 was installed for device %2, install time was %3 milliseconds.
161	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwareName" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_HiHighPartNew" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_LoHighPartNew" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_HiLowPartNew" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_LoLowPartNew" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_HiHighPartOld" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_LoHighPartOld" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_HiLowPartOld" inType="win:UInt32" outType="xs:unsignedInt"/> <data name="Prop_LoLowPartOld" inType="win:UInt32" outType="xs:unsignedInt"/> </template>	Software %1 was not newer, Version: '%2.%3.%4.%5'. Current Version: '%6.%7.%8.%9'.
162	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwareName"	Software %1 failed installation with error %2.

	inType="win:UnicodeString" outType="xs:string"/> <data name="Error" inType="win:UInt32" outType="win:HexInt32"/> </template>	
163	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwareName" inType="win:UnicodeString" outType="xs:string"/> <data name="Error" inType="win:UInt32" outType="win:HexInt32"/> <data name="ExitCode" inType="win:UInt32" outType="win:HexInt32"/> </template>	Software %1 failed installation with error %2 and process exit code %3.
164	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwareName" inType="win:UnicodeString" outType="xs:string"/> <data name="Error" inType="win:UInt32" outType="win:HexInt32"/> </template>	Software %1 had non-critical error %2 during installation, will retry later.
165	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwareName" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_CommandLine" inType="win:UnicodeString" outType="xs:string"/> </template>	Software %1 is being launched with command line: '%2'.
166	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DeviceInstanceId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_SoftwareLinks" inType="win:UnicodeString" outType="xs:string"/> </template>	Device %1 is requesting the following link from the Store: '%2'.
167	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwarePfn" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_ProductId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_IsFramework" inType="win:UInt32" outType="xs:unsignedInt"/> </template>	Product for pfn %1 located: ProductId: %2, IsFramework: %3
168	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_SoftwarePfn" inType="win:UnicodeString" outType="xs:string"/> <data name="ErrorCode" inType="win:UInt32" outType="win:HexInt32"/> </template>	Uninstalling existing pfn %1 failed with error %2
169	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ProductId" inType="win:UnicodeString"	ProductId %1 is already installed and is being checked for updates

	outType="xs:string"/> </template>	
170	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ProductId" inType="win:UnicodeString" outType="xs:string"/> </template>	ProductId %1 is being installed
171	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ProductId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_InstallType" inType="win:UnicodeString" outType="xs:string"/> <data name="ErrorCode" inType="win:UInt32" outType="win:HexInt32"/> </template>	ProductId %1 done processing, install %2 completed with error %3
180	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ProductId" inType="win:UnicodeString" outType="xs:string"/> <data name="ErrorCode" inType="win:UInt32" outType="win:HexInt32"/> </template>	Retrieving Store entitlement for pfn %1 failed with error %2
200		A connection to the Windows Update service could not be established.
201		A connection to the Windows Metadata and Internet Services (WMIS) could not be established.
202		The Network List Manager reports no connectivity to the internet.
203		The Network List Manager reports connection to the internet has been established.
220	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_NotificationHandler" inType="win:UnicodeString" outType="xs:string"/> </template>	Registered the handler %3 for the app %2 to handle notifications from the device container %1.
221	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	A handler for the app %2 was already registered for the device container %1.
222	<template xmlns="http://schemas.microsoft.com/win/2004/08/events">	The device container %1 and the app %2 specify

	<data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	background task information, but we failed to register with error %3.
223	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	Unregistered for the Print background task after uninstalling the app %1.
224	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	Unregistered for the Mobile Operator background task after uninstalling the app %1.
230	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	Made the request to the store to download the app %2 for device %1.
231	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	Successfully installed the app %2 from the store for device %1.
232	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	There was an error trying to install app %2 from the store for device %1. This operation will be retried when the device is reconnected or the user logs on again.
233	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_PackageId" inType="win:UnicodeString" outType="xs:string"/> </template>	There was an error trying to install app %2 from the store for device %1.
234	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_MilliSeconds" inType="win:Int64" outType="xs:long"/> </template>	Device %1 has had a driver update installed. Install time was %2 milliseconds
300	<template xmlns="http://schemas.microsoft.com/win/2004/08/events">	The device container '%1' has entered the ready state

	<data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> </template>	
301	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> </template>	Device setup for container '%1' has been completed.
302	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_ServiceInfoNamespace" inType="win:UnicodeString" outType="xs:string"/> <data name="Prop_CultureCode" inType="win:UnicodeString" outType="xs:string"/> </template>	Device metadata that contains an extension namespace has been parsed for container '%1', ExtensionNamespace = '%2', Culture = '%3'
400	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> </template>	
401	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:GUID" outType="xs:GUID"/> </template>	
402	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> </template>	
403	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	
404	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_PackagePath" inType="win:UnicodeString" outType="xs:string"/> </template>	
405	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_PackagePath" inType="win:UnicodeString" outType="xs:string"/> </template>	
406	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> </template>	

407	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	
408	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> </template>	
409	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_ContainerId" inType="win:UnicodeString" outType="xs:string"/> </template>	
410	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> </template>	
411	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> <data name="HRESULT" inType="win:UInt32" outType="win:HexInt32"/> </template>	
412	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> </template>	
413	<template xmlns="http://schemas.microsoft.com/win/2004/08/events"> <data name="Prop_DevnodeId" inType="win:UnicodeString" outType="xs:string"/> </template>	
7710		ENTER: Searching WU for driver updates.
7711		EXIT: Searching WU for driver updates.
7712		ENTER: Downloading driver update from WU.
7713		EXIT: Downloading driver update from WU.
7714		ENTER: Installing driver update from WU.
7715		EXIT: Installing driver update from WU.

Table 31: Event IDs generated by the ETW provider FCBB06BB-6A2A-46E3-ABAA-246CB4E508B2 (Microsoft-Windows-DeviceSetupManager)

References

Allievi, A., Ionescu, A., Russinovich, M., & Solomon, D. (2021). *Windows Internals, Part 2.*

ERNW_TMFW. (n.d.). SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package TMFW.

ERNW_WP2. (n.d.). SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 2.

ERNW_WP31.1. (n.d.). SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 31.1.

ERNW_WP4. (n.d.). SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 4.

Hale, J. (2015, February 4). *Digital Forensics Stream*. Retrieved from Leveraging the DeviceContainers Key: https://df-stream.com/2015/02/leveraging-devicecontainers-key/

Microsoft. (2021, 12 15). Adding a PnP Device to a Running System. Retrieved from https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/adding-a-pnp-device-to-a-running-system

Yosifovich, P., Ionescu, A., Russinovich, M. E., & . Solomon, D. A. (2017, May 3). *Windows Internals* (7th ed., Vol. 1). Microsoft Press. doi:978-0735684188

Yosifovich, P., Ionescu, A., Russinovich, M., & Solomon, D. (2017). *Windows Internals, Part 1: User Mode (Developer Reference).*

Keywords and Abbreviations